1970

A simulation and diagnosis system incorporating various time delay models and functional elements

David Michael Rouse

Department: Electrical and Computer Engineering

A SIMULATION AND DIAGNOSIS SYSTEM INCORPORATING VARIOUS TIME DELAY MODELS AND FUNCTIONAL ELEMENTS

by

DAVID MICHAEL ROUSE, 1945

A DISSERTATION

Presented to the Faculty of the Graduate School of

UNIVERSITY OF MISSOURI-ROLLA

in partial fulfillment of the requirements for the degree

DOCTOR OF PHILOSOPHY

in

ELECTRICAL ENGINEERING

1970

ABSTRACT

The application of digital simulation to all phases of digital network design is considered here as oppossed to development of simulation for one or two restricted parts of the digital process. For this reason a simulator is presented which can be consistent by varying the level of expression from the simulation of architectural structures to such detailed simulation requirements as race analysis of asynchronous sequential circuits.

In order to make system simulation more than just an idea, it must be capable of handling large circuits in reasonable times. It is demonstrated that functional simulation has the potential to increase simulation speed while reducing the required storage. This potential is realized with the following features of this simulator structure: 1) a modular structure for specification and execution, 2) the capability of being easily interfaced with gate level simulation, 3) the capability of utilizing the highest level of expression for simulation, 4) a variable level of expression, 5) a relatively unrestricted type of logic that can be simulated, 6) the capabilities of using standard functional modules, 7) a fairly universal means of expressing functional modules and, 8) the use

of data and control signals to further force selective trace capabilities on a module level.

Greater gate level simulation capabilities are obtained by extending the basic simulator to perform the simulation of undefined signal values and the simulation of ambiguities in signal propagation speeds.

The simulator presented here is part of a Test Generation and Simulation System. This system includes preprocessing, combinational test generation, automatic fault insertion as well as simulation.

ACKNOWLEDGEMENTS

I would like to express my appreciation to Dr. Szygenda for his enthusiasm for and constant supervision of this research.

I would like to acknowledge Dr. Tracey not only for the detailed review of this dissertation, but also for the guidance given me throughout my graduate degrees.

I wish to thank Jeanne for her understanding and sacrifice freely given while obtaining my degrees.

Appreciation is also extended to the National Science Foundation for the support given for this research under NSF Grant GK-2017 and under an NSF Fellowship.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Continued)

APPENDIX

LIST OF TABLES

CHAPTER I
INTRODUCTION

A. Past and Future Applications of Digital Simulation

Logic simulation is the process of trying to determine the operation of a digital network under given conditions.

Logic simulation was first used by the logic designer. This was an attempt, by visual inspection, to determine if the proposed logic design would function as expected. This is sometimes referred to as manual simulation. As digital networks became more complicated, manual simulation became less practical. The only alternative at the time was physical simulation which implies implementing the network to determine whether it functions correctly[1].

As digital computers become larger and faster, simulation of digital networks by digital computers becomes practical. Digital simulation had the advantage over physical simulation in that it could be used for check-out prior to network implementation.

As the advantages of simulation became more apparent, it was put to another use[2]. The objective here was to determine how a digital network would behave it faults existed in the network. This was an attempt to be able to readily detect faults that occurred in the implemented network. Simulated signal values were fixed to LOGICAL 1

or LOGICAL 0 to simulate a fault which acted similarly. This is referred to as fault insertion or fault simulation.

There exists[1,3-8] a number of digital simulators which have been implemented to do the simulation tasks as just described. However, due to the close association with the gate elements, the insufficient storage and speed of digital computers and inaccurate modeling, these simulators have been greatly handicapped[9-10]. Due to these limitations, which will be discussed in detail in later chapters, digital simulation has not been able to realize its full potential.

Some of the capabilities that lie in the grasp of digital simulation can be recognized by considering simulation as a tool of the complete digital design phase, instead of considering it only with regard to logic design check out.

Considering simulation throughout the complete design phase implies the use of digital simulation as a tool from design conception to implementation, including software structuring as well as hardware structuring. Thus, digital simulation could be a continuing tool which would make total system simulation, from start to finish, a reality instead of a dream.

These potentials and their possible realization will be the topics dealt with in this dissertation.

B. A "Big Picture" Approach to Digital Simulation

Digital systems are generally structured on a block or module basis, at least conceptually, since this is more consistent with man's grouping, or categorizing, thought process. Thus, design structuring is a process of defining modules, their function, and how they interrelate. Design is then, idealistically, the design of each of these modules.

Following the outline just indicated, for simulation to be a tool consistent and meaningful to all phases of design, it too should have the same form. Thus, simulation should first be structured in terms of modules, their function, and their interconnection. Then each module should be simulated at a gate level to determine its interaction as a module with the total system. Thus, modules could not only be designed in parallel, but could also be simulated in parallel in a system environment.

This "big picture" approach to digital simulation will be referred to as system simulation.

C. Terminology Required for Concise Expression of System Simulation

Existing simulators have been involved with the simulation of gates. It can be readily ascertained that the gate was the particular physical device being

simulated. The gate type indicated a particular group of gates, all having approximately the same physical parameters. The logical gate type referred to the logical operation that this group of gates performed.

If one generalizes upon the definitions above so that they can refer to things other than discrete gates or even discrete physical devices, the following definitions can be made:

1) An element is a device which corresponds to a definite entity during simulation and is not, during simulation, simulated as a collection of smaller elements. Conversely, a module is a collection of elements and can be simulated as such a collection of elements.

2) Element type indicates a group of elements having approximately the same parameters which describe it.

3) Element function type refers to the evaluation procedure of an element type.

Thus, a module which represents a collection of gates, such as an adder, could be simulated by an add instruction. Therefore, this module could also be represented by an adder element. The element type might be ADDQZ53 which

had certain parameters such as delay times, number of outputs, etc. The element functional type would be ADD which indicated the operation this element performed.

D. Simulation Study

An engineering study was performed on digital simulation and its application[11,12]. Many of the simulation objectives indicated in the previous sections were arrived at as a result of this study. However, areas other than just simulation[13,14] were involved in this study. These were combinational test generation[15], preprocessor, and post processor functions[16]. The total project involved test generation and simulation and was thus given the name TEGAS.

The area of simulation itself involved discrete time simulation, indeterminate value simulation, ambiguity region simulation as well as functional simulation. Discrete time simulation (Mode 1 simulation) will be discussed in Chapter II since it is the basis of the other three aspects of simulation. Indeterminate value simulation (Mode 2 simulation) as well as ambiguity region simulation (Mode 3 simulation), race analysis simulation, are also covered in Chapter II due to their similarity to discrete time simulation. Functional simulation will then be investigated in Chapter III. A

comparison and conclusion of these types of simulation will then be given in Chapter IV.

Chapter II

The Basic Simulator Structure

A. Desirable Simulation Feature

In the previous chapter the desirable features for the TEGAS system were indicated. These features imply certain features for the simulator itself which will be considered in this chapter.

A primary goal of this simulator is that it possess the ability to simulate any of the common modes of logic operation. The handling of asynchronous sequential circuits presents the most difficulty, in that circuit timing must be accurately described.

The speed of simulation (compilation and execution) is an important measure of a good simulator. Therefore, maximum speed is also an objective. However, the requirement for maximum speed can, and should, be sacrificed for more accurate simulation results, when required. Therefore, the speed of this simulator can be a function of the level of detail required of the simulation.

Another simulation goal is the use of a minimum amount of storage. The importance of this goal is two-fold. First, minimization of storage requirements will enable the simulator to handle larger circuits; and second, its use will not be limited to large computing

facilities. Appropriate segmentation can help reduce the storage requirements, but only with a sacrifice in speed. For these reasons, both minimization of storage requirements and software system modularity are of primary concern.

Machine independence is a desirable feature in order that this simulator not be limited to any particular computing facility. In addition to proper program structuring, a machine independent language must be used for all critical segments of the simulator. The simulator should be as flexible and versatile as possible, but at the same time it should be easy to understand and implement. Therefore, one need only be concerned with those options which are pertinent to the task at hand. For example, the option of simulating faults is available with or without race analysis.

It is necessary that, in order to use the speed and storage advantages of functional simulation, it must be compatible with gate level simulation. Thus, functional elements should appear the same as gate elements. No special consideration should be made during simulation to distinguish between functional elements and gate elements. This goes along with the desirability of easily adopting new element types without extensive modification of the existing structure. Therefore, this method should not require a large amount of reprocessing.

This flexability also implies an ability to easily add new unusual elements such as elements which simulate faulty elements.

B. Means of Achieving the Desired Features

There are two approaches that can be taken for digital simulation. One is the approach of modeling the entire circuit as one unit and, therefore, with one macro-model. This would be the technique used for a compiled simulator, since a compiled simulator levels and transforms the circuit into a form that can be dealt with collectively. The other approach is that of modeling the entire circuit by breaking it into smaller blocks, which can be individually modeled according to their type. This approach can be accomplished with a table driven simulator. A table driven simulator deals directly with elements, in that the circuit description is explicitly specified during simulation. Therefore, it determines, during simulation, what elements are to be evaluated next and then uses one generalized routine to evaluate all elements of any one type.

To provide the maximum versatility in adoption of new elements, the table driven simulator is the least dependent upon the element being simulated and deals with each element individually during simulation. Also a compiled simulator structure would require changing

the preprocessor (compiler) phase of the simulator. A table driven simulator also increases the similarity between gate level simulation and functional simulation in that since a table driven structure simulates elements independently then no distinction need be made according to the complexity of the evaluation procedure.

Not only must functional elements be compatible and consistent with gate elements but one must also be capable of changing from one type of description to another, without disturbing the overall description. This is achieved with the use of boundary elements.

A boundary element is an element which is placed at each input and output of a module to isolate that module from the rest of the circuit. Each boundary element has one input and one output, and the output value equals the input value. Taking this approach permits changes in the module definition without changing descriptions outside the module. This is depicted in Figure 1, where B1, B2, B3, and B4 are boundary elements for module M2. Gate G1 fans out to G2 and B1 when module M2 is expressed functionally. But, if module M2 is expressed at the gate level, without boundary elements, then the fan out of G1 would be to G2, G3, and G4. Without boundary elements, it would be necessary to change the fan out description of G1 (which is outside the module), if the user wants to change M2 to a gate

representation. But, with the boundary elements inserted, the fan out of G1 (to G2 and B1) is still the same independent of the type of expression used for M2. This could be of extreme importance, when a number of design groups are using the same high level description for the complete system, while considering their own particular section at the gate level.

Simulation is a process of evaluating signals of the logic net. In order to do this one must determine the order in which this evaluation is to take place. This is referred to as leveling in that, for combinational logic, gates would be organized into levels in which all inputs to a gate in one level were generated in a previous level. A leveling procedure of this type is sometimes referred to as space leveling, in that ordering is based upon the number of gates "deep" a gate is in the circuit. Most compiled simulators do space leveling for simplicity. A second approach would be that of time leveling in which ordering is done based on the time depth of a gate in a circuit. Time depth would then be the time required for an input change to effect the output of a particular gate. Time leveling is not necessary for combinational circuits since signal propagation times do not effect the steady state output value. However, time leveling is important for sequential circuits since signal propagation times can effect the steady state output value.

For this reason, one must time level to be able to perform correct simulation of sequential circuits. Or, one must make certain assumptions about the operation of the circuit, such as that it is not effected by signal propagation speeds.

Two other uses of leveling are encountered in conjunction with simulation. These are static leveling and dynamic leveling. Static leveling implies evaluation ordering prior to simulation, whereas dynamic leveling refers to evaluation ordering which occures during simulation. Again for combinational logic, and logic which can be represented as combinational logic, the simpler approach, static leveling is sufficient. A compiled simulation approach is most generally leveled in this manner. Table driven simulation is dynamically leveled in that, ordering is determined during simulation according to the element and circuit description tables. It should be noted that dynamic leveling can be accomplished by compiled simulators. But it is felt that this would make the compiling phase of simulation, an already cumbersome process, too data dependent.

With respect to the above considerations, simulation of sequential circuits would be more easily obtained in a table driven structure.

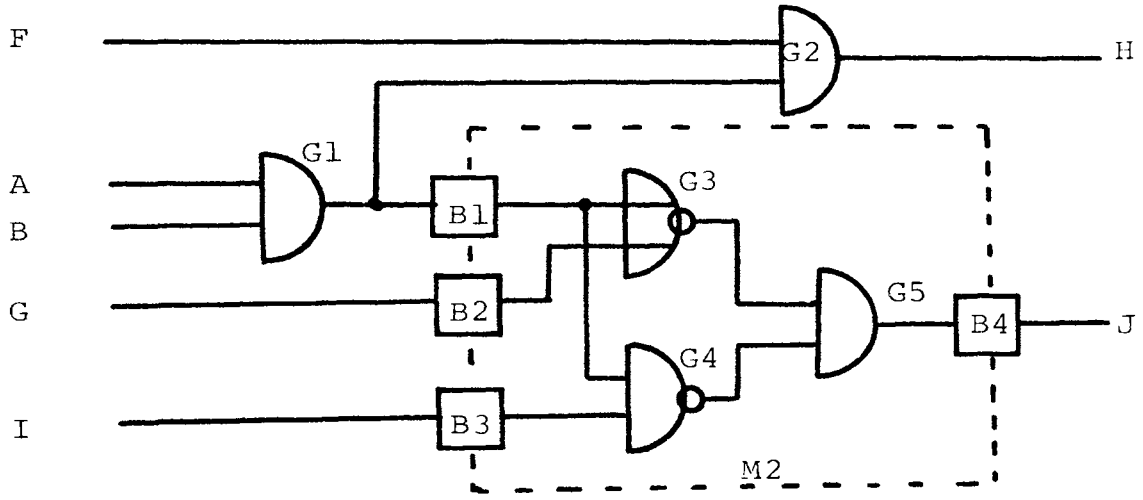


Figure 1: Module Boundary Elements

The ability to simulate asynchronous circuits relies on the ability to accurately represent the time associated with evaluation of signal values. By including the propagation time of each element in its description, and then using this parameter during simulation to order the evaluation procedure, this time factor can be accurately represented. This means that one must have the ability to accept propagation delays of different lengths instead of making a unit delay assumption. The former can be implemented in a table driven structure quite easily but would be quite difficult in a compiled simulator since compiled simulators are most generally space leveled instead of time leveled.

In many simulators, simplifying assumptions are made (such as considering only combinational circuits) as a result of the particular cases at hand or in order to obtain the greatest speed or least storage requirements. Thus, in order to simulate circuits which do not conform to these constraints, it must be assumed that the circuit function properly under these constraints. This is evidenced in the Seshu sequential analyzer[3] in which sequential circuits are simulated under combinational constraints. Thus, one must make a special case out of feedback lines. For this reason another side feature of a table driven simulator is that of not being required to consider feedback lines by special, cumbersome, and inaccurate methods.

These methods mentioned above include explicit specification of what lines are feedback lines and the ability to reset these to any value. These two conditions are produced when an attempt is made to represent an entire sequential circuit by one model, as is done in a compiled simulator structure. The latter case, which is commonly referred to as a reset assumption, is avoided since simulation occurs directly from a given accessible state without requiring reinitialization of the state during simulation.

One important feature of a table driven approach, with respect to race analysis, is that race analysis occurs concurrently when more than one race is active. Therefore, only one simulation must be performed for n nested races, as compared to as many as 2^n simulations for other approaches.

As mentioned earlier, speed and storage will be a primary concern in this simulator. Three techniques will be employed to reduce these problems to an acceptable level. They are: (1) selective trace simulation[2], (2) parallel simulation[20], and (3) functional simulation.

Selective trace is a technique used in conjunction with table driven simulators which provides the ability to evaluate only those elements which have a potential of changing. For example, one need not reevaluate a gate output if all the input signals are the same as they

were when it was last evaluated. Thus, simulation becomes a process of tracing changes, and their effects, through the circuit.

Signal values can be stored in one of three manners. First, one bit of a machine word could be used to represent the value of a signal. Second, each bit of a machine word could be used to represent a different signal value. Third, each bit of a machine word could represent different values of the same signal for different input vectors or different fault conditions.

The first of these three techniques is extremely inefficient in storage handling. The second approach is hard to execute in Fortran (the implementation language for the system) since it would require bit manipulation. Therefore, the third approach was taken. For this technique, n different input vectors (where n is the number of bits in the machine word length), or fault conditions, can be simulated with the same speed and storage required for the first approach. This is referred to as parallel simulation, since n unique simulations occur in parallel. The effect of this approach is to divide the required simulation time by a factor of n.

Another important implementation feature is called functional simulation. This is the grouping of a number of logic elements together and then expressing the group by its function[17]. Thus, one need only store and evaluate the function in order to simulate the represented logic. An example of functional simulation would be the representation of an adder by storing and executing a simple add instruction, instead of storing and executing the large number of logic elements used to form an actual adder circuit. Therefore, it can be seen that functional simulation enhances simulation speed and reduces storage. The ability to implement functional simulation is compatible with a table driven simulator structure, since it models the circuit by modeling elements, regardless of the evaluation procedure used to model the element. For this reason, changing or adding element types is a simple task which involves changing only the evaluation procedure and its respective pointer.

Fault insertion is also simplified since it now becomes a matter of simply providing elements having the same characteristic as a faulty element.

When faults are automatically inserted, fault collapsing is used to reduce the number of possible faults. This is done by inserting only one fault of a group of faults which always produce the same outputs

for any input combination. For example all stuck-at-0's on the input of an AND gate appear the same as a stuck-at-0 on the output of that gate. Therefore, the stuck-at-0's on the input need not be simulated if a stuck-at-0 on the output is simulated, since they all produce the same response and are therefore repetitious.

The foregoing considerations lead to the conclusion that a table driven structure would be the best approach for the simulation problem, as outlined in the previous chapter. To rephrase the more detailed features of the simulation previously indicated would be to describe it as a table driven simulator using parallel and selective trace simulation techniques along with the ability to do high level simulation.

Such a simulator has been implemented and a detailed description of it is given in Appendix A. A simplified description is included in the following section in order to insure a basic understanding of the operation of the system being described.

C. The Basic Simulator Software Structure

The first major implementation decision was the choice of a programming language in which the simulator would be written. Assembler, Fortran and PL/1 were considered. Utilizing an assembler language could result in a little faster execution, with somewhat less storage

required. However, Fortran was chosen since it would be easier to implement and is considerably more machine independent. Although PL/1 has some seemingly desirable features, they were sacrificed for the more commonly acceptable Fortran and the small decrease in execution time and storage. It was also desirable for this simulator to be acceptable for use on smaller machines, which have limited storage and compiler facilities. For these reasons Fortran was considered more desirable than PL/1.

The basic simulator consists of three tables, the Time Queue Table (TQ), the Circuit Description Table (CDT), and a table which contains the Current Value of each signal (CV). The Time Queue Table contains events that occur at time t, where t is the index of the Time Queue Table. The Circuit Description Table contains pointers to the evaluation routines used to determine the output values of the element, pointers to the fan in and fan out, and also contains the number of fan outs for each signal.

Using these three tables, simulation is performed as follows:

1. All values that exist in the Time Queue, at the current simulation time, are transferred to the Current Value Table, thus causing any projected changes in value to take effect.

2. If the new value entered in the Current Value Table is different from the old value, then all elements that are immediately affected by this change are reevaluated. (This is accomplished by following a fan out list).

3. The results of these reevaluations are projected into the time queue at the current time plus the nominal propagation delay of the signal.

4. The current time is incremented until an entry is found in the Time Queue, and then the process is repeated again.

This process is restated in a flow chart form in Figure 2.

In addition to this basic structure, other tables are used for optimization of both speed and storage. For example, functions are evaluated indirectly, via the Function Description Table (FDT), which also specifies additional parameters used in the evaluation routine. These parameters are: function type, time delay, number of inputs, and bus length. This permits a minimum number of evaluation routines that must be provided for simulation. By use of the FDT table, the same routine

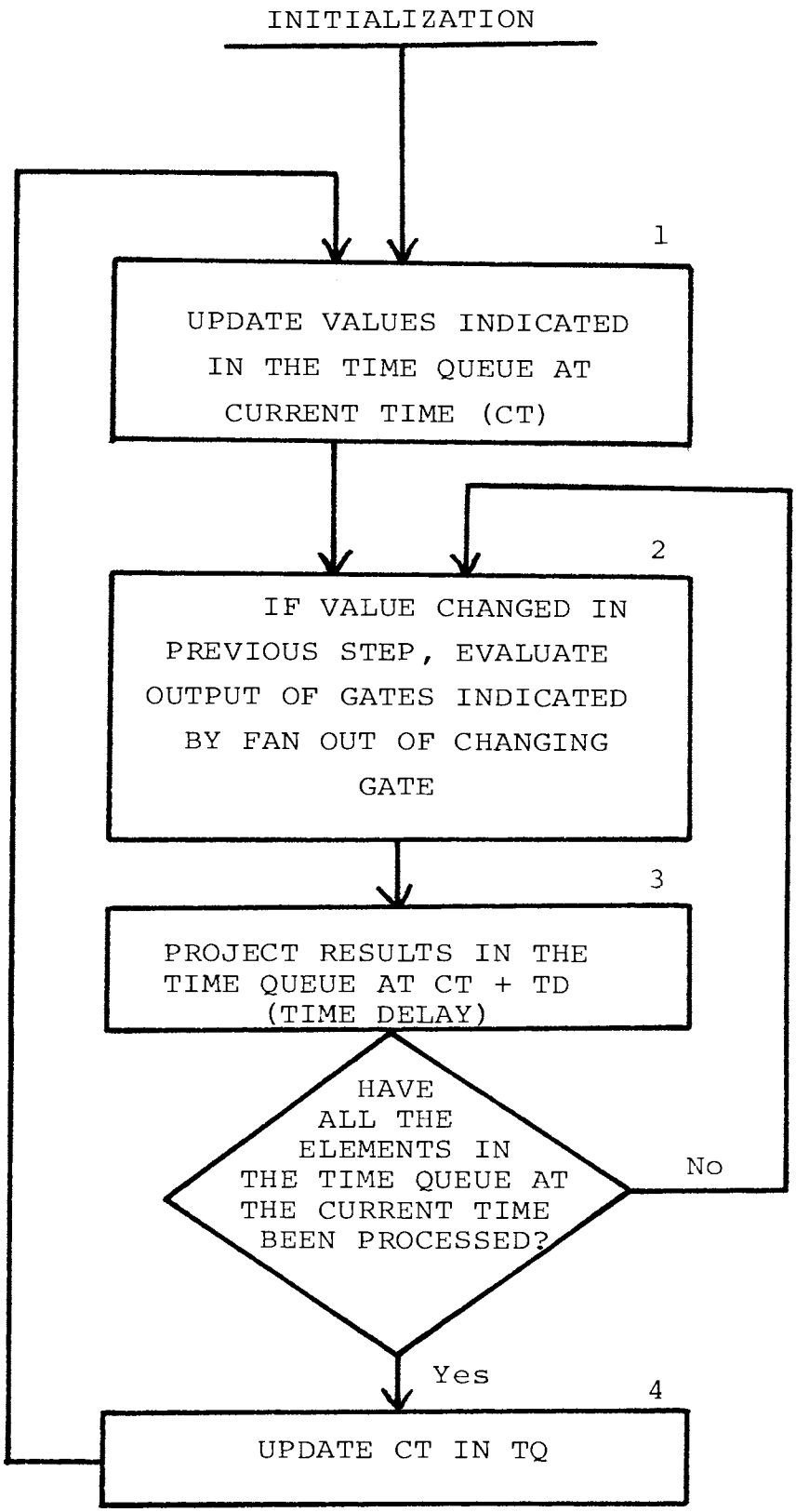


Figure 2: Flow Chart of Simulator

would be used for a 2 input AND gate with a time delay of 5, as would be used for an 8 input AND gate with a time delay of 8.

To keep the size of the TQ from becoming prohibitively large, a Macro Time Queue table (MTQ) was implemented to store events which occur at large time intervals, relative to the largest propagation delay for any gate. The Time Queue was then made cyclic in coordination with the MTQ, where each cycle of the TQ advances the MTQ one step.

Some gate level elements, such as flip-flops, as well as functional elements, have multiple outputs. To be able to simulate this type of element, an additional entry is provided in the CDT Table, which is used to chain the output together.

A goal of this system simulator was to possess capability of simulating elements other than actual gates, such as functional modules. Since functional modules are just as apt to be dealing with busses as with single lines, the ability to specify bus lines collectively would make functional module specification an easier, as well as a more meaningful task. For this reason, the capability of specifying busses collectively has been implemented with the use of a paging scheme, where Bus Value (BV) is a group of pages and Bus Value State (BVS) is a table which indicates use, length and

location. Therefore, the CV of a bus in an indirect pointer to a page, which contains the actual values of the bus signals.

D. Mode 2 - A Three Value Simulation

Three value simulation is a simulator which contains a third value to indicate whether a signal is known at any instant of time[18]. This type of simulation will be referred to as Mode 2 simulation.

Mode 2 simulation thus contains three values (1,0,I) to represent the conditions of a known logic 1, a known logic 0 and an indeterminant value. An AND gate which had a 0 and I as inputs would have a 0 input. The output of an AND gate which has a 0 input is always 0 regardless of the other inputs. The output for an AND gate with inputs of 1 and I is I. A table for evaluation of 2 inputs AND and OR gates is given in Table 1.

The obvious application of Mode 2 simulation is the propagation of unknown signals to determine their effect upon circuit performance. A specific application would be its use for determining whether a given initialization vector (a subset of the total state) was sufficient to properly initialize the circuit.

Implementation of Mode 2 simulation from a Mode 1 simulator is an extremely simple task. This involves

AND	0	1	I
0	0	0	0
1	0	1	I
I	0	I	I

a. Logical AND

OR	0	1	I
0	0	1	I
1	1	1	1
I	I	1	I

b. Logical OR

Table 1: Logical Table for Gate Evaluation in a Three Value Simulation

providing storage for the third value and implementing the logical operations indicated in Table 1. It is due to this simplicity that Mode 2 simulation is so attractive.

The only question that is of any consequence is that of the exact value assignment of 0, 1, and I. Two choices are available. The first is to use a separate bit to indicate the validity of the actual signal value which meant compatibility of representation with Mode 1 simulation. The second approach would be to use a less obvious assignment but which would produce less complicated gate evaluation routines. Two such assignments would be as follows:

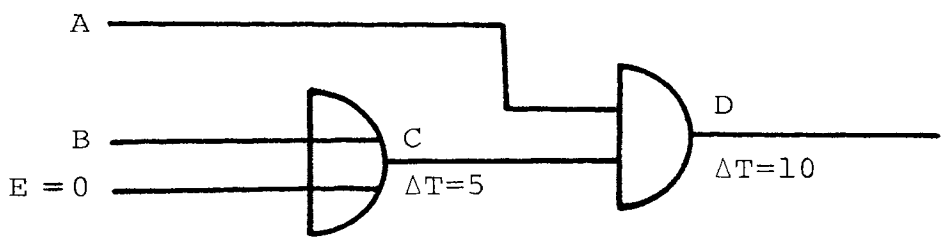
Code	Meaning
00	logical 0
01	logical 1
10	Indeterminate
11	Indeterminate

First Assignment

Code	Meaning
00	logical 0
01	Indeterminate
10	Indeterminate
11	logical 1

Second Assignment

The first representation was chosen for compatibility reasons. An example of the Mode 2 simulation is given in Figure 3. Here, each signal can either be 1, 0, or I (I indicates Indeterminate). Before time 0, all signals are unknown and, therefore, in an indeterminate (I) state. At time 0, A and B are changed to a 1. As a result of this change in A and B, the value of C and D must be



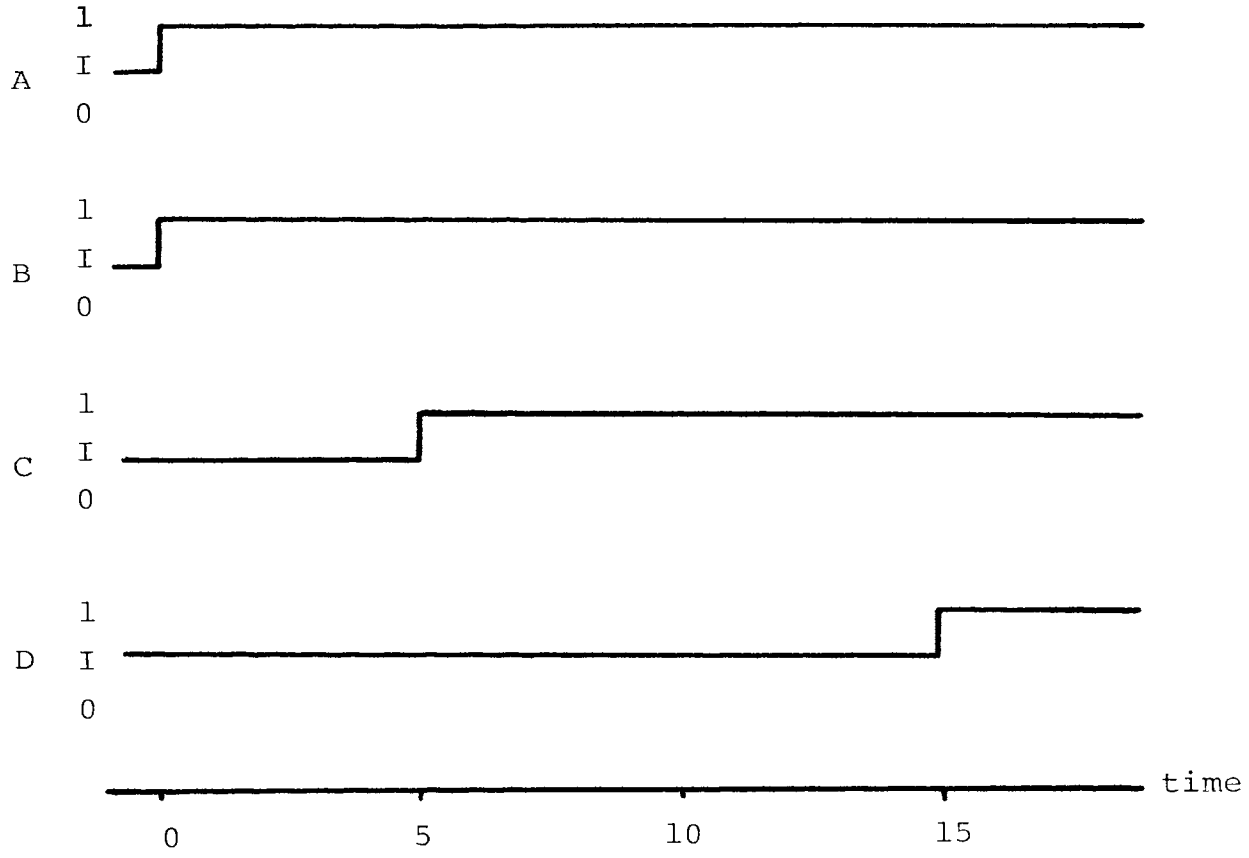


Figure 3: Mode 2 Simulation

reevaluated. C is evaluated to be 1, at t = 5. The change in A, at t = 0, causes a reevaluation of D. However, since C = I at t = 0, then D = I at t = 10 due to the change in C having not yet propagated to D. An evaluation table for C and D is indicated in Table 1. However, the change in C at t = 5 causes D to be evaluated again. Hence, D becomes 1 at t = 15 since both C and A are known.

E. Mode 3 - A Race Analysis Simulator

To further increase the accuracy of simulation, an ambiguity interval can be associated with each signal. This is a result of an inability to specify exactly when a given signal will actually make a transition from one state to the next. The requirement of an ambiguity interval comes from the fact that gates of the same type could have different propagation times. Therefore, the time delay of a gate would be represented as a minimum value plus an ambiguity region. By considering this ambiguity, race and hazard analysis can be performed during simulation. This type of simulation is referred to as Mode 3 simulation.

In general, the mode 3 simulator is used to propagate ambiguity regions to provide determination of the existance of essential hazards. The unique characteristic

of the Mode 3 simulator, that does not exist in the other modes, is that it carries regions instead of single values. A technique similar to this has been described by D.L. Smith[19]. A spread is depicted with the use of a New Value (NV), as well as a Current Value (CV), along with the Potential Error Value (PE). During simulation, this ambiguity region is represented by simulating the earliest possible transition point (the CV) and the latest possible transition point (the NV). The potential error is then a logical function of the CV, NV, and the PE. With respect to the simulator, this evaluation procedure simply appears as another element type.

The tabular relationship between the CV, NV and PE for the inputs of a two input AND and OR gate are given in Table 2. Since a parallel simulation approach is taken in this simulator the evaluation of PE is accomplished by the following logic expression:

AND GATE

$$PE = (NV1 + PE1 + CV1) \cdot (NV2 + PE2 + CV2) \cdot (\overline{CE2} + PE1 + \overline{CV1} + PE2) \cdot (\overline{NV1} + PE2 + PE1 + \overline{CV1})$$

OR GATE

$$PE = (\overline{NV2} + PE2 + \overline{CV2}) \cdot (\overline{NV1} + PE1 + \overline{CV1}) \cdot (NV2 + PE2 + NV1 + PE1) \cdot (PE2 + CV2 + PE1 + CV1)$$

LOGICAL AND

CV1, PE1, NV1

CV2, PE2, NV2	000	010	011	001	101	111	110	100
000	0	0	0	0	0	0	0	0
010	0	1	1	1	1	1	1	1
011	0	1	1	1	1	1	1	1
001	0	1	1	0	0	1	1	1
101	0	1	1	0	0	1	1	0
111	0	1	1	1	1	1	1	1
110	0	1	1	1	1	1	1	1
100	0	1	1	1	0	1	1	1

PE

LOGICAL OR

CV1, PE1, NV1

CV2, PE2, NV2	000	010	011	001	101	111	110	100
000	0	1	1	0	0	1	1	0
010	1	1	1	1	0	1	1	1
011	1	1	1	1	0	1	1	1
001	0	1	1	0	0	1	1	1
101	0	0	0	0	0	0	0	0
111	1	1	1	1	0	1	1	1
110	1	1	1	1	0	1	1	1
100	0	1	1	1	0	1	1	0

PE

Table 2: Logical Table for Potential Error Evaluation in Mode 3 Simmulation

The input values of lead one are CV1, PE1 and NV1 and those for lead two are CV2, PE2 and NV2.

The functions for CV and NV are the logic functions of the particular gate relating CV1 and CV2 and NV1 and NV2, respectively.

In order to extend Mode 1 to Mode 3, three things must be done. First, the original evaluation procedures must be replaced by those mentioned above. Secondly, storage must be provided for PE values and NV values. Lastly, the destinction between CV transitions and NV transitions must be made in order to propagate these changes correctly. This last distinction is necessary since the first transition (transition of CV) has a delay time of just the minimum propagation time whereas the last transition (transition of NV) has a delay time of the minimum propagation time plus the ambiguity time.

An example of the Mode 3 simulation is depicted in Figure 4. A and B are input signals of initial value 0 and 1, respectively. C and D are the output of inverters, which have a propagation delay of 4 and an ambiguity of 2. E, which is the set signal to an S-R Flip Flop, is the logical AND of C and D. At t = 0, A changes to 1 which produces a change in C to 0 through an ambiguity region from t = 4 to t = 6. This means that the change of C could occur sometime between t = 4 and t = 6. If B changes at t = 1 (possibly due to an ambiguity in the

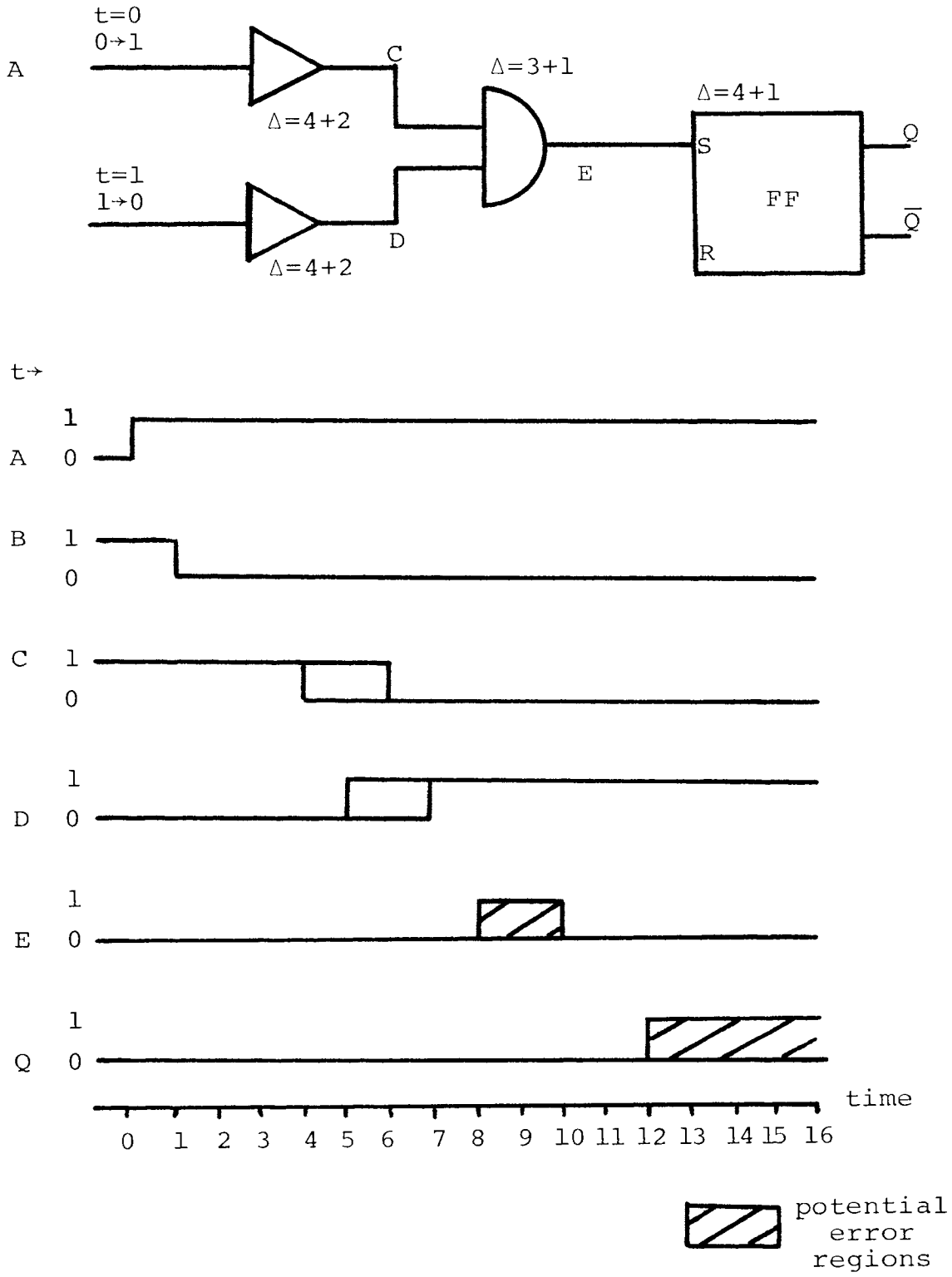


Figure 4 : Mode 3 Simulation

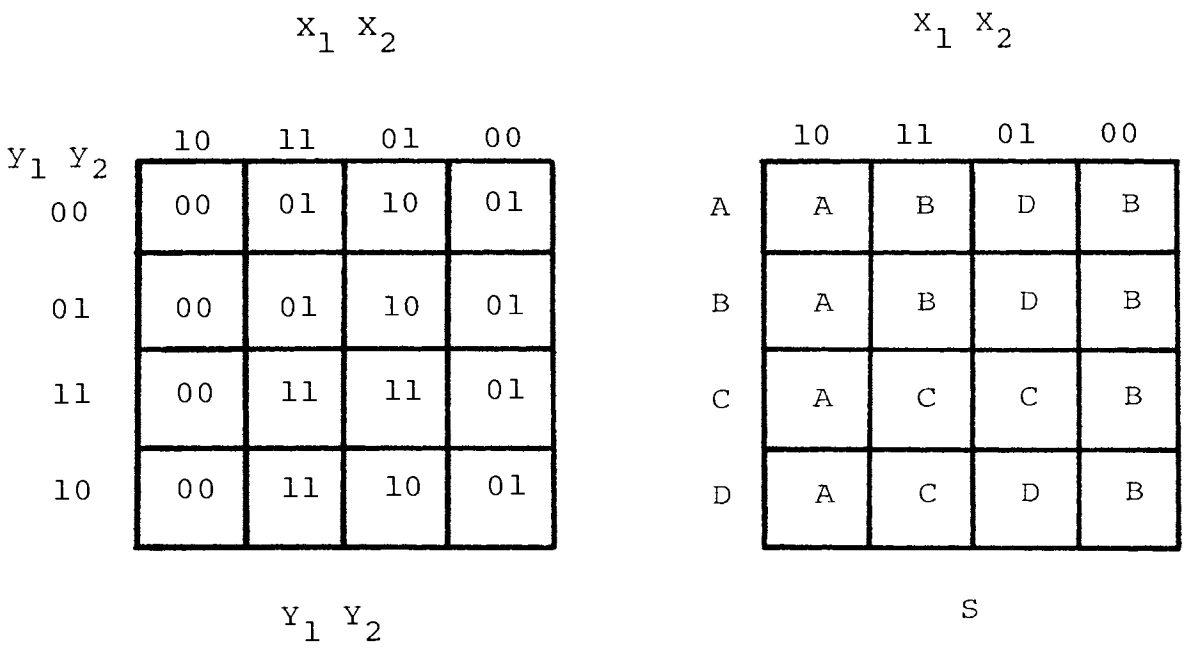
circuit feeding B), then D changes value as indicated. Notice that the value of E, as a result of C and D being 1 between t = 5 and t = 6, is a potential error region. This, along with ambiguity and minimum delay of the AND gate, produces the results indicated for E. Since E is setting the Flip Flop, this potential error region sets the Flip-Flop to a potential error value, which is the resulting state of Q. From this example, it can be seen how the ambiguity in propagation delay is handled, as well as how Mode 3 simulation handles potential error regions and how these potential error regions can detect race conditions.

Sequential logic circuits containing global feedback loops are extremely difficult to simulate, even for the simplest of design philosophies. Whether accurate simulation is achieved, most often depends upon the type of sequential action used in the design which must be consistent with the particular simulator being used, or the person simulating the circuit having a very intimate understanding of the circuit operation. These two circumstances are more often the exception rather than the rule. In many design environments, these conditions are rigidly forced upon the user by their simulation structures. In order to alleviate this problem, the largest possible user flexibility was a goal for this simulator. One of

these degrees of freedom is presented in the following example, which shows how race analysis of an asynchronous sequential circuit can be performed.

A transition table and state table are given in Figure 5 for a simple sequential circuit which has been implemented in Figure 6. A race can be seen to exist between states C and D for the input vector 01 and stable state B. Whether this race is a result of improper design, characteristics of a circuit containing a faulty element, or an intentional risk, is unimportant. The important thing is that the simulation of this circuit is capable of revealing sufficient information to determine how the circuit will, or could, act when physically implemented. Figure 7 shows the response of X_1 changing from a 1 to a 0 according to the unit delay assumption (the circuit is in stable state 01, with X_1X_2 = 11). From Figure 7, it can be observed that the circuit makes a transition from state B to state D, a seemingly definite and satisfactory result.

By considering the circuit response as indicated in Figure 8, which uses more accurate delay time information (as given by the minimum delay in Figure 6) for each gate, it is observed, through this type of simulation, that all isn't as simple as indicated by the previous simulation. It can be seen that, as the accuracy of



$$Y_1 = \overline{X}_1 X_2 + y_1 X_2$$

$$Y_2 = X_1 X_2 + \overline{X}_1 \overline{X}_2 + \overline{X}_1 y_1 y_2$$

Figure 5: A Transition Table and State Table for a Simple Sequential Circuit

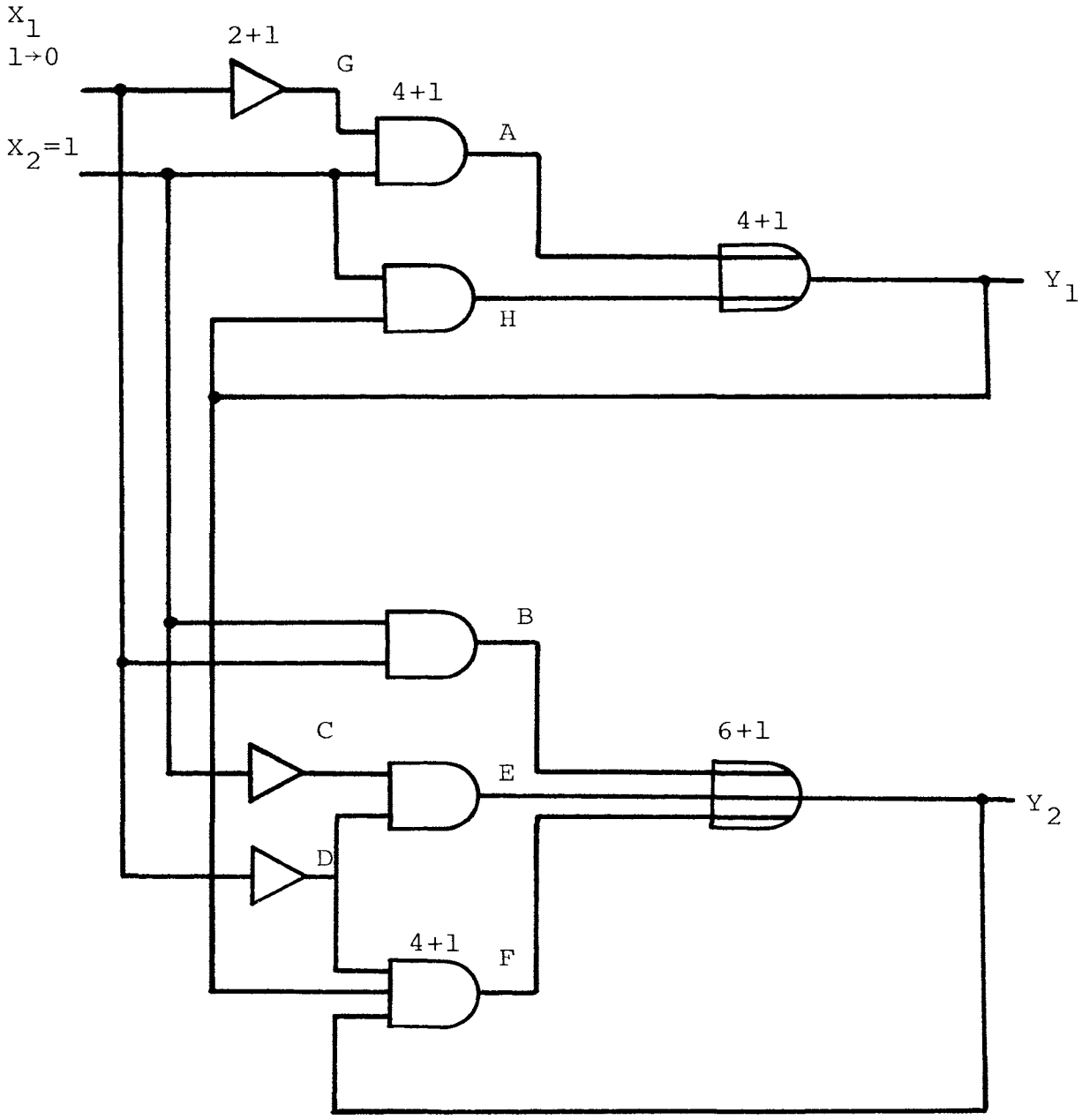


Figure 6 : Example Sequential Circuit

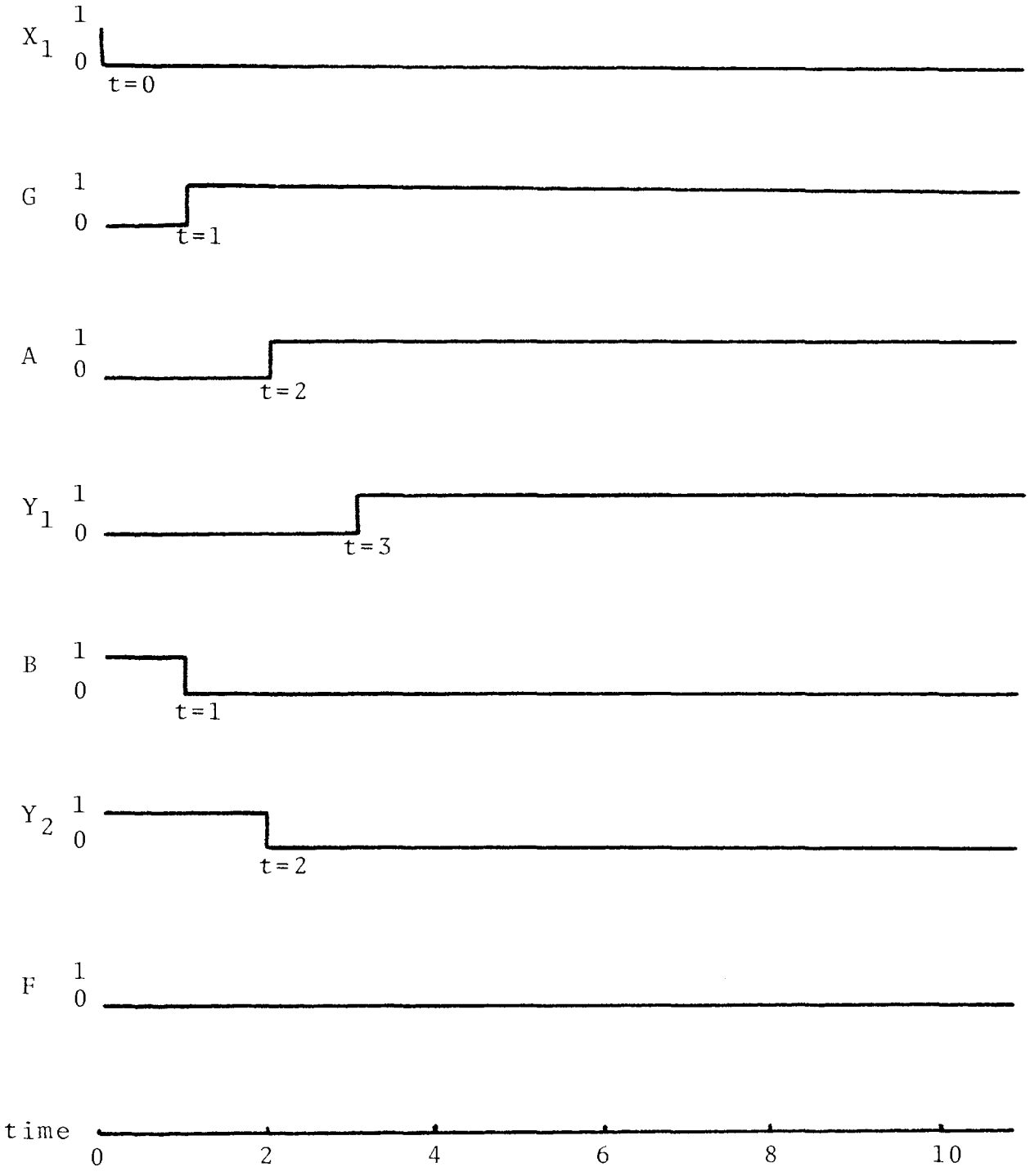


Figure 7: Results for Circuit with Unit Time Delays

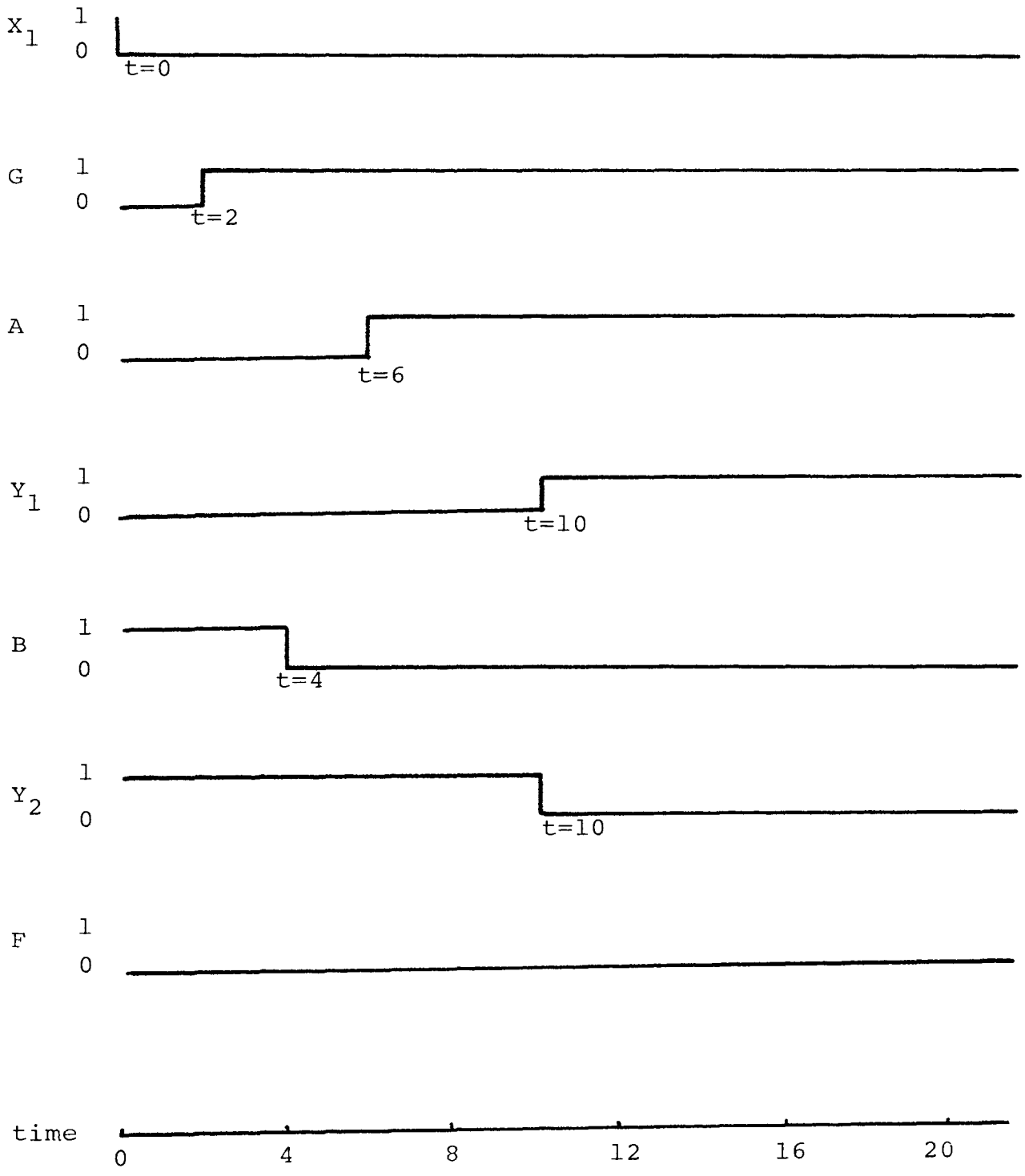


Figure 8: Results for Circuit with Variable Time Delays

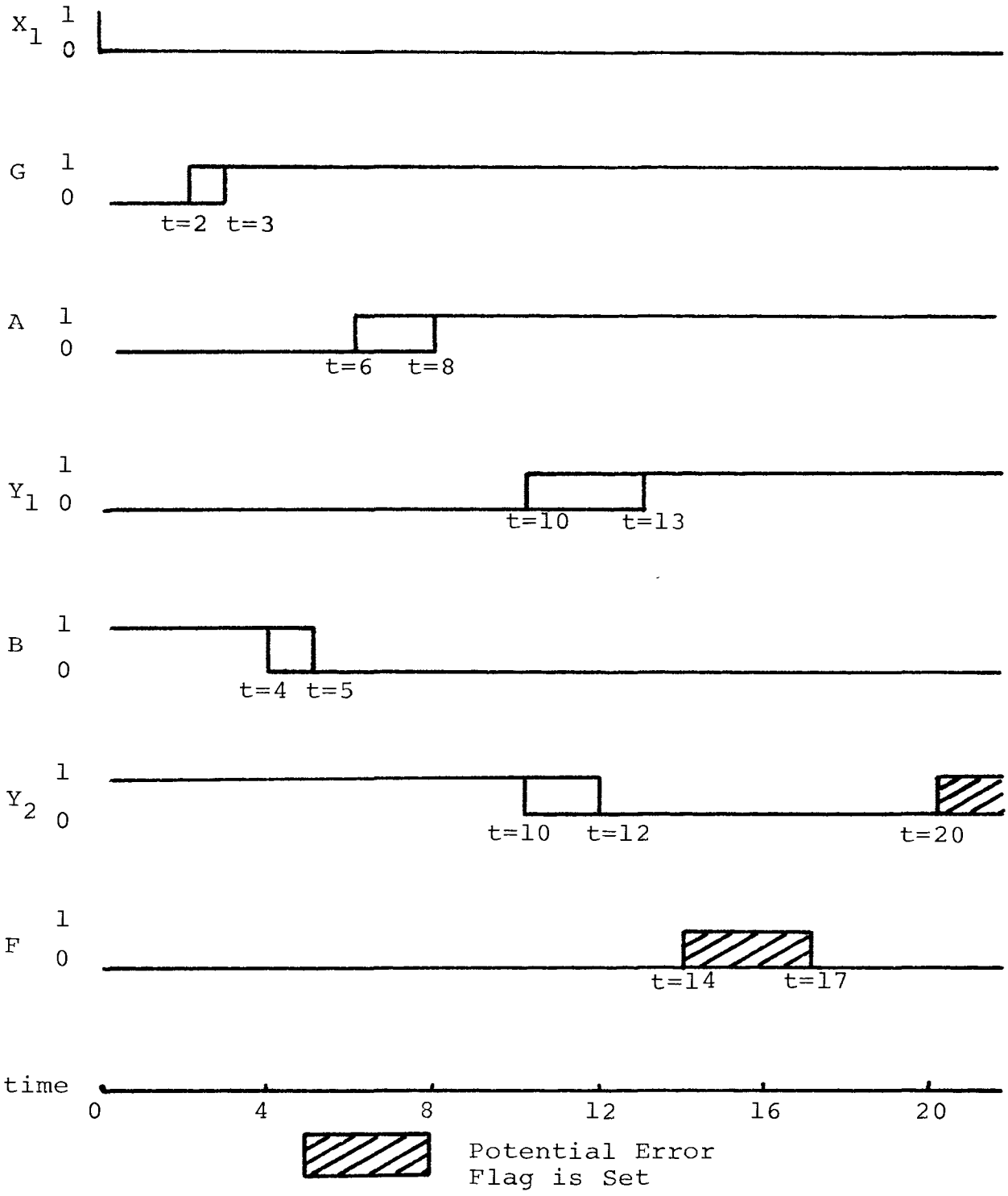


Figure 9: Results for Circuit with Variable Time Delays and Ambiguity

propagation time becomes closer to that of the physical circuit, the race between states D and C becomes apparent. It appears here as the simultaneous transition of Y_1 and Y_2. This type of simulation is one which might be performed by Mode 1 simulation.

The curiosity raised by using a more accurate representation for the delay time can be satisfied by also considering an ambiguity time (as indicated in Figure 6 and 9) associated with each gate. The state variable Y_1 could change anywhere between t = 10 and t = 13. This is a result of the possible variation in time delays of the inverter, AND gate, and OR gate, along the propagation path X_1. Similarly, Y_2 could change between t = 10 and t = 12. The value of F is essentially the AND of Y_1 and Y_2, since $\overline{X}_1$ appears as a constant 1. However, the AND of the two ambiguity regions for Y_1 and Y_2 is not only another ambiguity region in F, it is also a potential error region as well. In actuality, F may or may not produce the momentary 1 spike between t = 14 and t = 17. Note that a transition region is concerned with the question of <u>when</u> a transition will take place. However, a potential error region is concerned with <u>whether</u> a transition could take place.

Thus, from this example, it can be seen that a variation is propagation delay is enough to produce a

critical race condition from a seemingly stable design, wrongly as indicated by a unit delay simulation. This condition is detected in Mode 3 simulation when the potential error flag is set for the state variable Y_2, as indicated by the shaded area in Figure 9.

Chapter III

Functional Simulation

A. Introduction

The synthesis and analysis of large digital systems can be divided into three phases; logic design, diagnostic test generation and software development.

In many instances, these tasks are treated as separate entities. Logic design is performed first, with the other phases following in a somewhat independent manner. A basic premise of this paper is that these tasks should be integrated. Only an integrated, parallel, approach can adequately satisfy the objectives of all three areas. This approach has not been practiced, in many cases, due to the lack of adequate techniques needed to integrate these phases.

One such technique, that could be utilized for all three tasks, is digital simulation of the system. While the problem of simulation is by no means new, existing techniques have been plagued with inadequacies. Two major difficulties, associated with many simulators, are the prohibitive storage requirements and unreasonable simulation running times, for large systems. These problems appear to be a direct result of simulators

being restricted to a gate level approach to simulation.

Gate level simulation is required for various portions of the logic design and diagnostic test evaluation sequences. However, these requirements do not necessitate gate level simulation of the entire system.

The synthesis and analysis procedure for large digital systems can be described in terms of the generalized sequence, shown in Figure 10. Design usually starts with the conception of an idea and a verbal description of the goals of the system (Block 1). This description must be transformed into a more precise description of how the goals are to be achieved (Block 2). The transformation generally consists of successively subdividing the system into subsystems. Each component subsystem is defined by subsystems interrelation and internal memory specifications. Design verification of the high level system is necessary before branching into the areas of logic design, software development, and diagnostic test generation (Block 3,5 and 7, respectively).

Ideally, these three functions should be initiated simultaneously, and developed concurrently. This not only produces the shortest design cycle time, but also permits inter-area feedback.

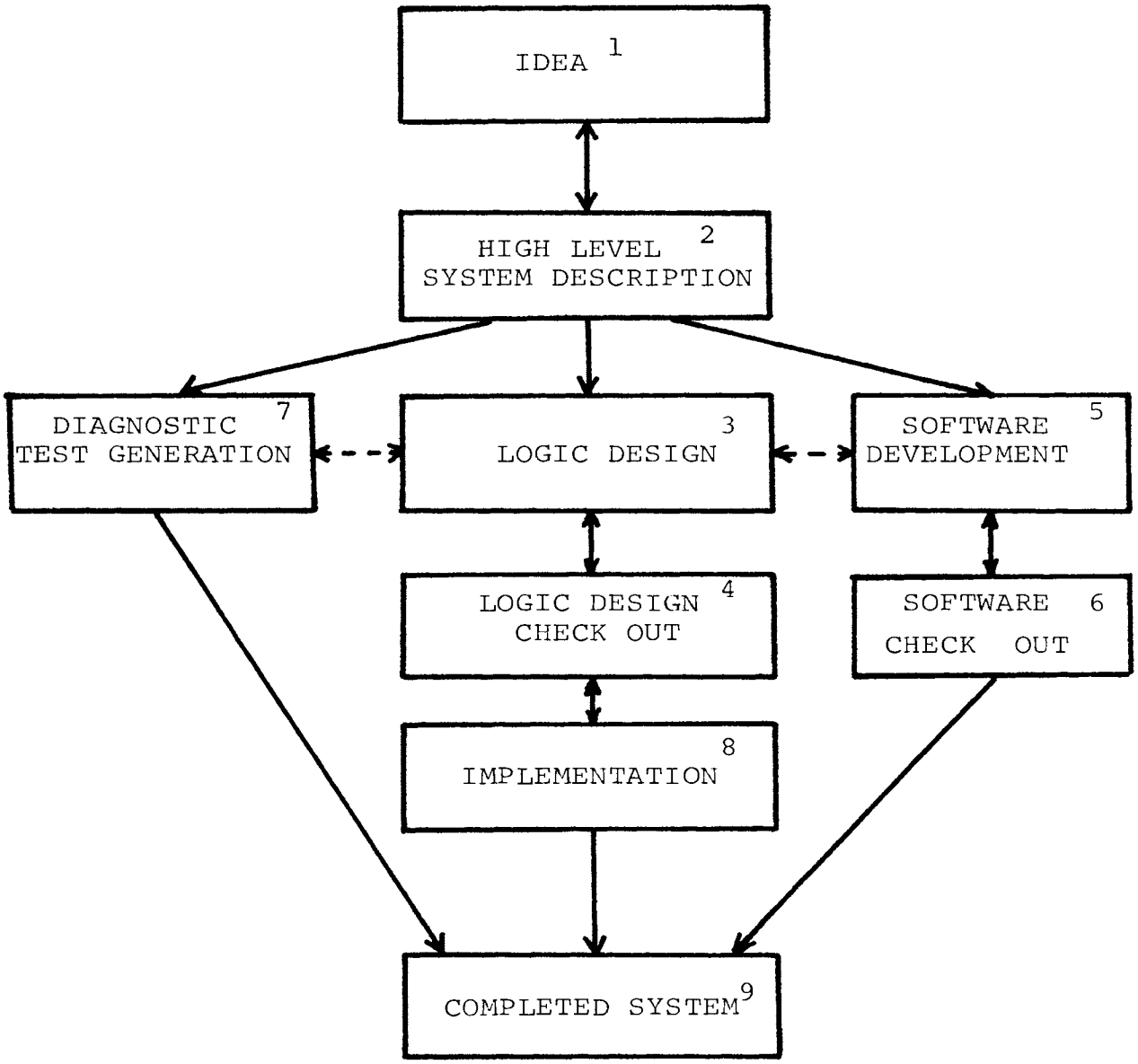


Figure 10: Design and Analysis Sequence for Large Digital Systems

The logic design and check out sequence (Block 4) expands the logic design equations, derived in Block 3, into a logical element form. At this level of design, it is necessary to verify the fulfillment of original design specifications.

Diagnostic test generation should also be started as soon as possible, preferably in parallel with logic design. In the past, diagnostic test generation has been initiated after logic design has been completed. This is understandable, to some extent, since test generation usually involves simulation at the gate level, hence, it must wait until portions of the system are at the logic level of design. However, preliminary results can lead to easy design modification for maintainability. Therefore, integration of design and diagnosis appears to be the most practical approach to the systems maintainability problem.

Ideally, diagnostic test generation should proceed concurrently with logic design, but it must also be considered with the total system when determining the system responses under fault conditions. This is identical to the problem of logic design check out. Hence, many of the problems that exist there, exist here also. One primary difference is that, in diagnostic test generation, a large number of simulation runs are necessary, making simulation speed a primary objective.

The software development sequence should parallel that of hardware development. Complete software check out (Block 6) is often neglected during early design stages. This results in crash programs for software verification and numerous undetected bugs which continually plague the users.

Therefore, software routine generation and check out should also be started as early as possible. But, since execution is necessary for check out purposes, software generation is often not completed until the actual system is produced. In the simulation of software, the macro-characteristics of the system are of primary concern. On the other hand, in logic design simulation, the micro-characteristics are of major importance. Thus, the question arises as to whether or not the same simulator should be used for both logic design check out and software check out. It has been demonstrated that since the logic design is being simulated on a system basis, only a small part of the total system, at any given time, is being simulated at the logic level. Thus, software check out could be accomplished by simulating the total system at the highest level (the same high level simulation that is being used for logic design check out and diagnostic test generation).

If a software simulator is not already available, then one must decide what type of simulator would best suit the job (which is generally not fixed, but dependent upon the particular design philosophies and requirements). This could involve simulation decisions between synchronous and asynchronous, compiled or table drive, sequential or purely combinational, etc.

Attempting to visualize some of the problems in trying to achieve this ideal design sequence provides some insight to the type of simulator needed. If the design specification of Block 1 and Block 2 are to be checked for consistency, then the simulator, employed for this check out, should be capable of simulating the design in a form consistent with, if not identical to, that used in the specification of Block 2. Therefore, the simulation models should be described (at least at this phase of simulation) by a form consistent with that of Block 2, such as, a universal design language and standard predefined functions for the more common modules.

For the logic design check out sequences (on a system basis) to function concurrently with design, the simulator should have the capability of simulating parts of the system at the gate level and the remainder at a functional level. This requirement establishes the need to express and evaluate modules by their functions, in

lieu of expressing and evaluating each logic gate of the module.

It would also be advantageous for the functional description to be the same description as that used to check out the high level systems design. From this, it can be seen that a variable level of expression is necessary in order to permit dynamic utilization of the highest level for simulation. This allows one to change the level of simulation without requiring a recompile phase of the total system.

In retrospect, the key to achieving an ideal design sequence is the ability to define logic modules such that they can be expressed collectively by the function they perform and consistently with the gate level function they represent. Thus, the key to large system simulation is functional simulation. Also, the desire to interchangeably simulate functional modules and gate descriptions produces an interfacing problem between the gate level simulation and functional simulation.

Gate level simulation occurs when a digital system is represented by the gates or logic element that are actually used in forming the machine. The digital system is thus simulated by evaluating each logic gate.

On the other hand, functional simulation is the grouping of a number of logic elements together and then expressing this group by its function. Thus, one needs

only to store and evaluate the function, in order to simulate the represented logic.

An example of functional simulation would be the representation of an adder by storing and executing a simple add instruction, as opposed to storing and executing the large number of logic elements that are used to form an actual adder circuit.

From this example, two advantages of functional simulation can be seen. These are increased speed and reduced storage.

These advantages are a result of making use of higher level instructions of the host machine (as the add instruction in the previous example).

Another not so obvious benefit of functional simulation is that of specification simplification. This is to say that it is a much simpler task to specify the element types of an adder module as being an adder, than it would be to describe it by giving its gate representation.

Functional simulation also acts as an organizer for simulation and simulation specification. For example, a number of similar signals (signals which originate from the same element and have essentially the same fan-out) can be grouped together and treated as one signal. Such groups of signals are referred to as busses. This type of organization not only increases simulation speed and

reduces storage but also simplifies the task of system description.

Another such organization technique is that of making the distinction between control signals and data signals. A control signal is one which could cause a change in the output signals of an element. A data signal is one which will not cause a change in the output signals of an element. Thus, when a signal changes, it need not fan-out to any elements where it is a data input, since such a signal change will not in itself change the output signal value.

In order to be able to make a smooth transition from one design step to the next, of the design sequence mentioned earlier, a variety of levels of expression are needed. This is because the type of specification most applicable to each design step is not the same. For example, the first phase of design may be performed in a register transfer language, whereas the logic design phase would be dealing with gate level specification. Between these two might be used large element packages such as adders or decoders.

Consistency between design steps help to identify some of the desirable types of functional elements as well as some of their common characteristics. One characteristic is that they all should appear as block type elements thus having a defined set of inputs, outputs and internal states. The types of functional

elements can be seen to fall into three classes.

1. non-standard functional elements which can be generated from a language or languages consistent with computer design.

2. standard functional elements which are library routines for commonly used blocks of gates.

3. gate elements which can be used in conjunction with functional elements either collectively or individually. Thus an element might be expressed by one gate or a large number of gates.

B. A Functional Level Example

In an attempt to establish firmly the concept of functional elements let us consider a small computing system.

A very small computing system might be depicted functionally, as in Figure 11. Here, the system is represented by four modules:

1. The Control Unit (CU)
2. The Instruction Interpretation Module (IIM)

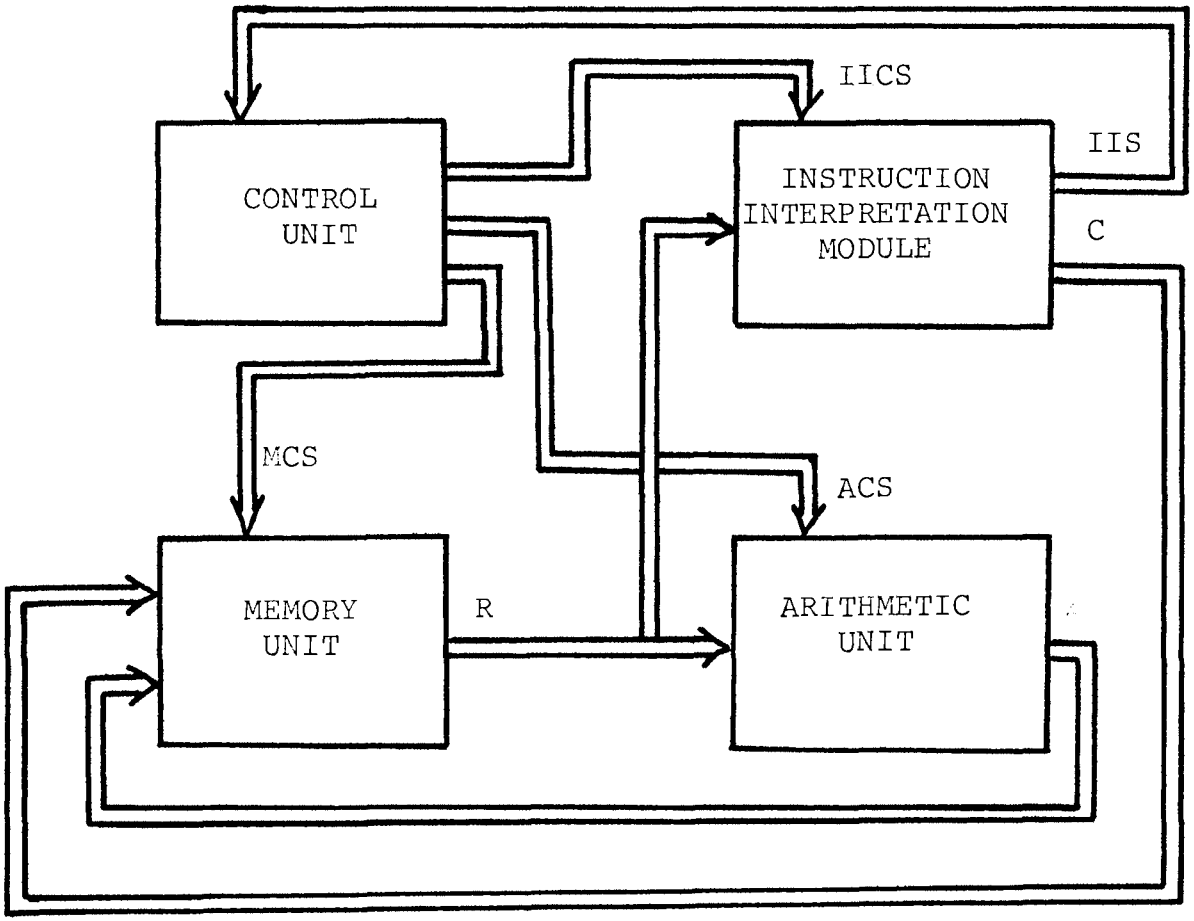


Figure 11: Modular Layout of a Small Computer

3. The Arithmetic Unit (AU)

4. The Memory Unit (MU)

The C register is the output of the Instruction Interpretation Module and it contains the address of the next memory location to be accessed by the memory module. The R register is the value that has been accessed by the memory module. The result of any arithmetic, or logical operation, is present in the A register as the output of the Arithmetic Unit. The Control Unit determines the sequence of actions of each of the other modules by issuing control signals through the control signals IIS, IICS, MCS and ACS. Thus, the execution of an add A to memory instruction, which is located in the R register, would be as follows:

1. The Control Unit would issue a signal to the IIM, indicating that an instruction is in the R register which is to be interpreted. Upon completion of the interpretation, the IIM presents the address of the operand in memory to the MU and the results of the instruction interpretation to the CU by the IIS lines.

2. The CU provides an appropriate sequence of actions according to the IIS value.

In this case, it would be to instruct the Memory Unit to access the contents of the location indicated by the C register.

3. After the contents of location C is present at R, the CU would issue a control signal (ACS) to the AU to perform an add of the A register and the R register, placing the results in A.

4. This being completed, the CU would then start the execution of the next instruction by instructing the IIM to increment the address of the present instruction and place it in the C register.

From this example, one can see that each module has a function to perform in determining its output value according to its input value.

In this system, a module is considered to be a collection of gates (possibly only one) which can be represented collectively, as a function.

In general, each module of Figure 11 can be represented by the general module, shown in Figure 12.

Thus, a system is modeled by segmenting the total system into modules, each in the form of Figure 12, where each module is represented by a model consisting of a

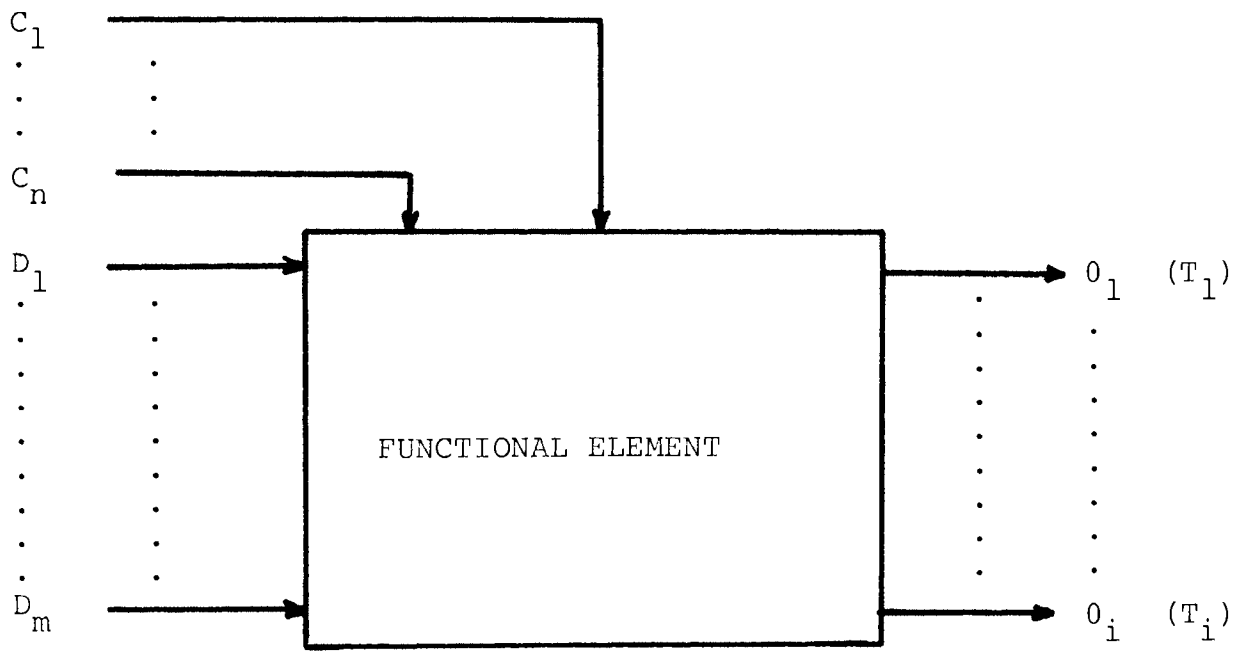


Figure 12: Functional Model

collection of inputs ($C_1 \ldots C_n$, $D_1 \ldots D_m$) and output ($0_1 \ldots 0_i$), corresponding to hardware signals, and a function relating these inputs and outputs. A time delay (T_i) is also associated with the outputs of the module. The time delay is the propagation time of a signal from the inputs to the output of the module.

The inputs are subdivided into two groups: data signals, $D_1 \ldots D_m$, and control signals, $C_1 \ldots C_n$. The distinction between the data and control signals is that a change in a data signal will not cause a change in the output value, but a change in a control signal could conceivably produce a change in the value of an output signal. This distinction plays a significant role in the simulator structure chosen, and it will be considered in greater depth in the explanation which will follow.

This model is given more versatility in Figure 13 by permitting the inputs and outputs to be busses. This is helpful in two respects: first, systems are generally initially specified in terms of busses, and secondly, this reduces the amount of intermodule connection specification required for system description.

Thus, the inputs and outputs of Figure 13 are either single signals or bus signals, which are a collection of single signals. Single signals are represented by a single arrow (→) while busses are represented by a double arrow (⇒). Therefore, signals C_1, D_1 and 0_1 are single

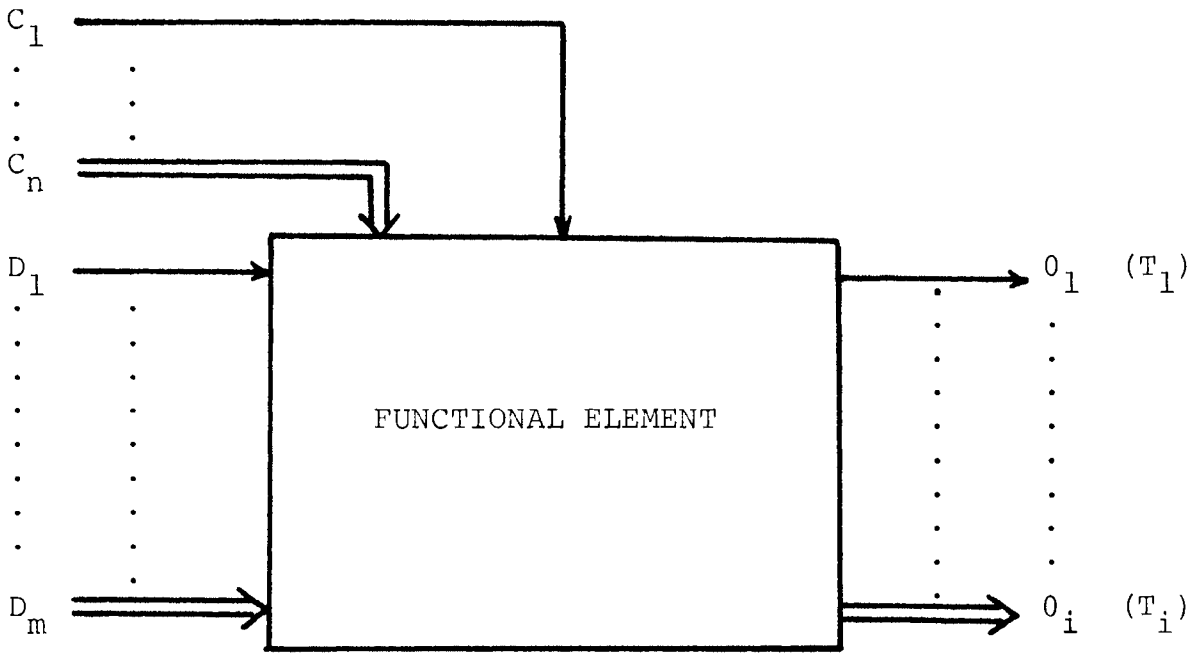


Figure 13: Functional Model with Register Bus Lines

signals and represent one value, while C_n, D_m and O_i are bus lines and represent more than one value.

C. Approaches and Techniques Used for Functional Simulation

Up to this point the desirable features of functional simulation have been discussed without consideration of how or whether they can be implemented. It is the intention of this section to discuss how these techniques for functional simulation are implemented. It will also be seen how some of them rely upon or enhance the implementation of others.

Busses are the representation of a collection of signals which have similar properties such as fan-in, fan-out and time delay. Thus, the description of a buss is identical to that of a single signal in that they both are expressed by giving a fan-in, fan-out and evaluation procedure. The distinction comes in the storage of the actual value that the signal or signals represent. Since a single signal is more often represented than a buss, then a buss signal should be an extention of a single signal representation. This extention is made by storing the actual values of this buss in a page and letting the value normally used for the single signal point to the beginning of this paper. This approach is very appealing in that in order to

manipulate the values of this buss, one need only change the pointer associated with the buss.

This paging approach to buss manipulation does, however, require a paging scheme to make efficient use of storage. This can be done by using a page directory which indicates page usage as well as page location and length.

In order to possess the capability of changing a module description to another form of description, a boundary element is used. For example, one time the description for an adder module might be a functional element, whereas the next time it may be desirable to express it at the gate level.

Another side benefit from boundary elements is that it gives a definite and single location of all signals which cross a module boundary. This can be used in conjunction with the implementation of data signals so that data signals can be readily changed to control signals when necessary. For example, a functional element could make effective use of data signals. If this module were simulated at the gate levels with faults present, these data signals should be simulated as control signals. This is a result of not being able to guarantee that, under fault conditions, data signals would not act as control signals since correct operation of the module is not necessarily the case.

Boundary element implementation is a convenient result of the module description process in that a module is described in terms of primary inputs and primary outputs. This implies a dummy element to indicate source and distination. These dummy elements are then later used as boundary elements when describing the total system in terms of modules with primary inputs and outputs.

The initial interest in functional simulation is its ability to simplify the evaluation procedure of an element. For example, the evaluation procedure for an adder module is much simpler using an add instruction than it would be using logical operations. This principle can be extended, with greater effect, to such less obvious applications as memory modules. A paging scheme can be used in such a module just as effectively as is done in an actual computer.

A number of such applications and their implementation is given in Appendix B, section II.

For functional simulation to become a usable feature, it must be both versatile and easy to express and use.

Ease and versatility of expression have been obtained by allowing five types of element expressions. These are:

1. Gate Element
2. Standard Functional Element
3. A Computer Design Language
4. Fortran
5. Compiled Gate Description

Elements can consist of a single gate. Thus, the evaluation procedure for a gate element would be specified by indicating the gate type, which indicates a predefined logical relation for gate evaluation. For example, Figure 14 gives part of the logic for the Control Unit (CU), referred to earlier. Here, each module is composed of one gate, and all input lines are control lines. Gate A would be evaluated according to its gate type, where the gate type would indicate the characteristic of the gate, such as logical type, time delay, and number of inputs. The logical type would indicate the logical relation used for output evaluation. The time delay and number of inputs would be parameters used in the output evaluation.

Standard functional elements are similarly predefined elements, which can be used by declaring a particular element to be a standard functional element. An example would be an N-bit 2's complement adder. Thus, to define a complete adder, one need only give its function type, the number of bits being added, and its time delay.

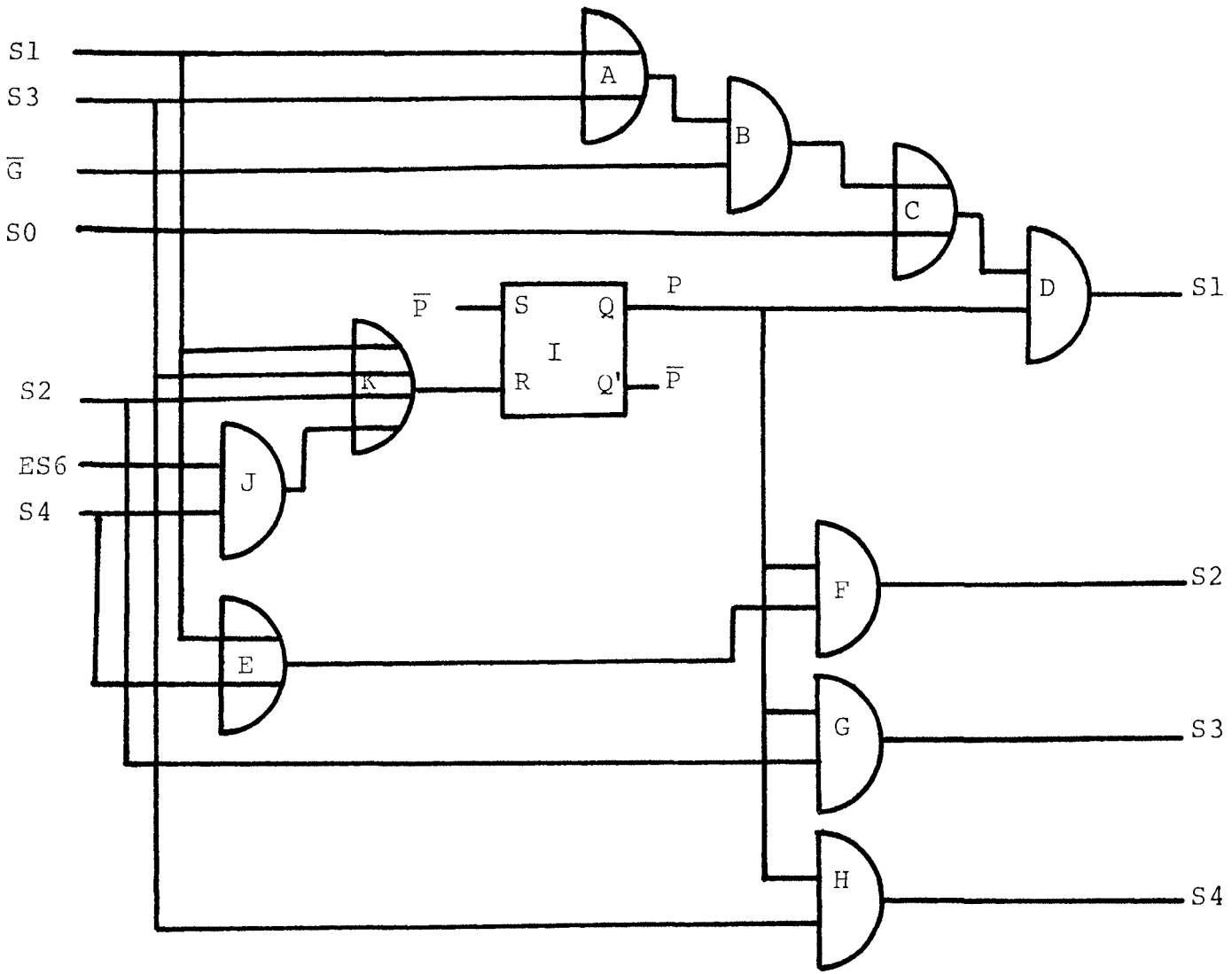


Figure 14: Control Unit Logic

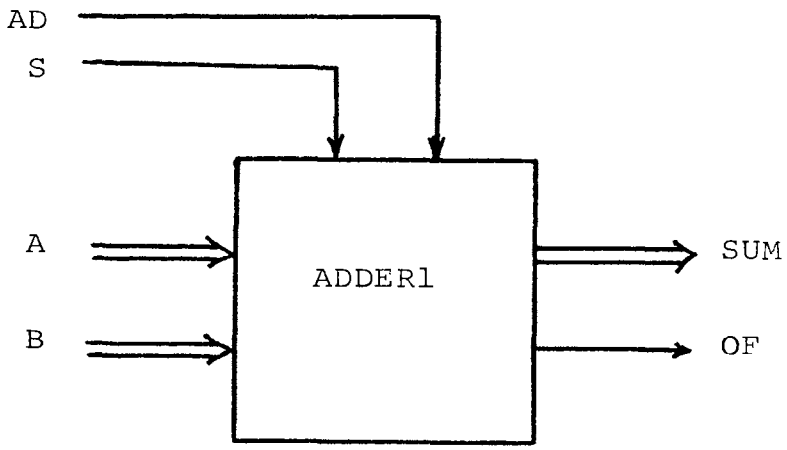


Figure 15: Adder Module

Figure 15 gives the pictorial representation of the ADDER1 element. It is defined by element type, the time delay, and number of bits being added. The element type indicates the routine used for output evaluation, the time delay and number of bits are parameters used by the routine. There is a strong resemblance between gate elements and standard functional elements.

To permit the initial design specification to become the initial functional representation for design verification, a computer design language[21] is allowed for module description. In order to provide sequential element control at a design language level, sequential elements can be described in a flow table form.

Often, it is desirable to check the logic of a design before it is actually committed to hardware. For this reason, functional elements can be expressed

in terms of a computer design language which is converted to Fortran for simulation, thus permitting the initial design specification to become the functional description for design verification. The computer design language makes use of two types of statements, conditional transfers and function calls. Conditional transfer statements are of the following form:

$$C1 \cdot S2: \quad A = B \cdot K3 + C \cdot K4$$

If C1 and S2 are true then A will be replaced by B·K3 + C·K4. A, B and C can be defined as registers. Any logical expression is permitted to the left of the colon and to the right of the equal sign. Function calls are of the form C1·S2: CALL ADDER (A,B,C). Any logical expression can exist on the left of the colon and any predefined system subroutine can be used on the right of the colon.

In order to be able to break a total system into functional modules at the design language level, sequential control elements can be used. These are described in a flow table form and can be used to generate the control variables which sequence the functional elements. Such a flow table description for a sequential element is indicated in Figure 16. S0, S1, S2, S3 and S4 are the states of the sequential machine and G is the condition variable. Therefore, if S4 is true upon entry

of the module, and G is false, then S1 would be returned true and S2-S4, S0 would be returned false.

	$\overline{G}$	G
S0	S1	S0
S1	S2	S1
S2	-	S3
S3	-	S4
S4	S1	S2

Figure 16: Flow Table Description of a Module

Fortran element descriptions can be used as a means of generating new, efficient standard functional elements. It can also be used to generate special elements, whose function cannot be described easily by a high level language. For example, it might be necessary to have a sign-magnitude adder. It would not only be slow, if written at a design language level, but would also be awkward and require unnecessary program storage.

Combinational gate level modules can be executed as an element, in a compiled fashion. That is, a number of gates can be expressed collectively as an element and then these gates will be converted into a compiled simulation structure. This feature is based upon the assumption that for certain types of modules

a purely compiled simulator structure would be capable of faster execution with less storage than a table driven simulator. When no functional description is available, the gate description could be used as a functional description.

The number of evaluation routines needed for a system can be greatly reduced by writing these routines in general and then specifying additional parameters according to the element type. Thus, the element type would indicate a particular type of element or a number of elements which have the same characteristics. Element function type refers to the evaluation procedure of the element. A number of element types might all have the same element function type. For example, a 2 input 4 ns AND gate and a 3 input 6 ns AND gate are of different element types but of the same element function type (AND). Such an evaluation structure requires fewer element function routines and therefore less storage.

Extreme care must be taken, however, when writing such routines not to make them so general that they are no longer effective for the element they were originally intended.

D. Steps Involved in Functional Simulation - An Example

In order to show the steps involved in functionally simulating a digital system, we will use the modular layout of the system, demonstrated in Figure 11. This figure provides the initial modular structure of the system.

From this figure, it can be seen that one can define the operation of the computer by specifying the operation of each module. The instruction interpretation module might now be described in a computer design language, as in Figure 17, which uses conditional logic expressions and function calls to define the operation of the module.

S1, S2, S3, S4 are major states of the system and JMP, JMPC and LCID are instruction commands. Thus, when the computer is in state S1 (S1=1), then C and D are set to zero. When in state S2, D is incremented by 1 by the subroutine INCD. When S3 is true, the address portion (ADDR) of the instruction is placed in C and the op-code portion (I) of the instruction is decoded into K by the DECODR subroutine. If S4 is true, and JMP or JMPC is true, then ADDR is placed in D, but, if LCID is true, D is placed in C.

```
S1:        C = 0
S1:        D = 0
S2:        CALL INCD (D)
S3:        C = ADDR
S3:        CALL DECODR (I,K)
S4•JMP:    D = ADDR
S4•JMPC:   D = ADDR
S4•LCID:   C = D
```

Figure 17: Instruction Interpretation Module In a Design Language Representation

After the total system has been logically checked at this level, then a more detailed representation for the IIM could be used. Such an expression is given in Figure 18 which shows the IIM at a register bus level, using standard functional modules as building modules.

As in the design language specification, if S3 is true, the I register is decoded into K. The D register is controlled by signals S1, JMP + JMPC, and S2. As before, if S1 is true, D = 0, if JMP or JMPC is true, D = ADDR; and if S2 is true, D = D1; where D1 is D incremented by 1. Similarly, the C register is controlled by S3, S4, LCID and S1. It can be seen from this example that gate elements can be used right along with functional units, since they are all depicted as modules.

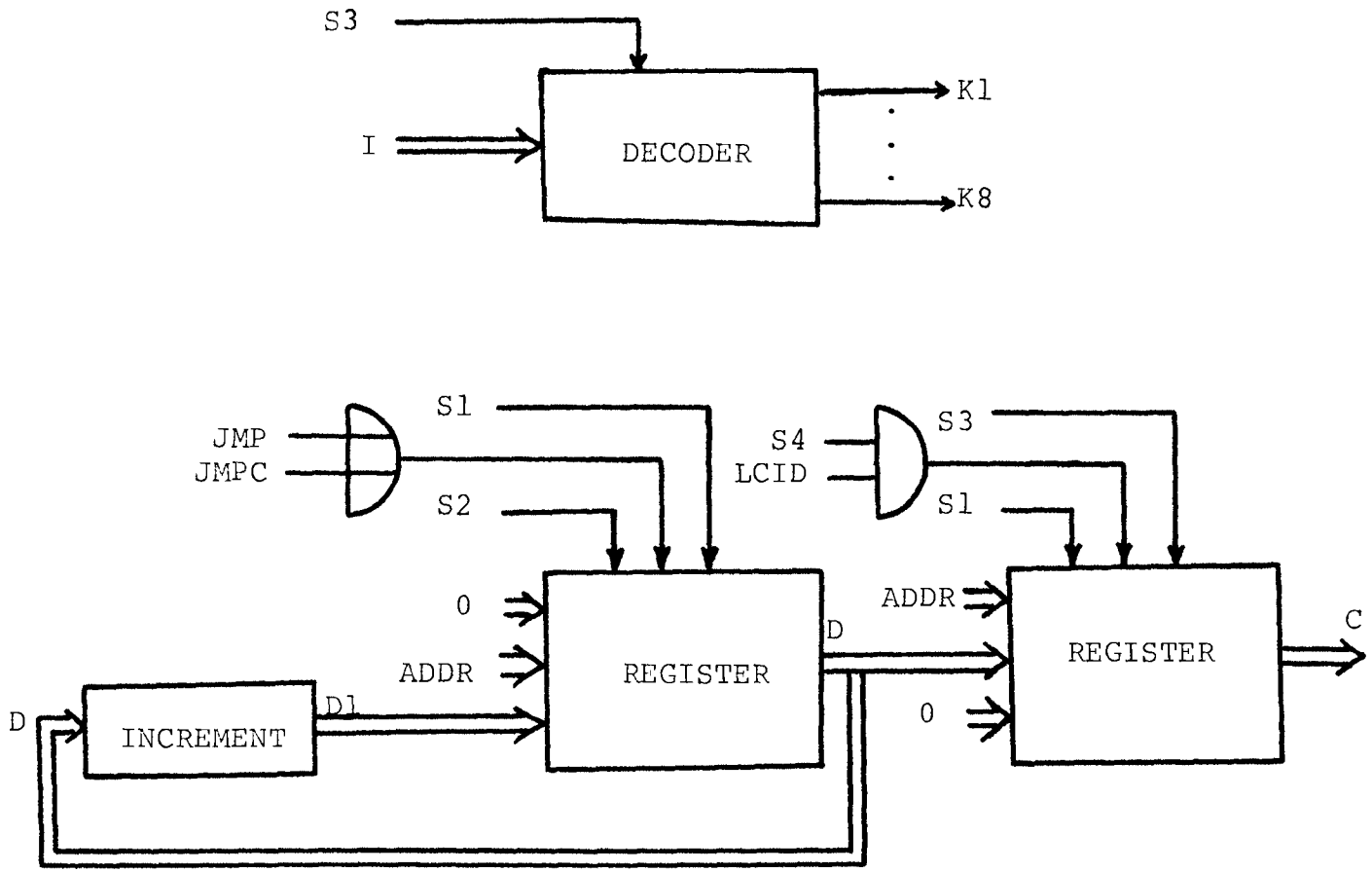


Figure 18: IIM at a Register Module Level

The standard functional module representation is better than the design language module not only in that it is closer to the actual hardware representation, but its execution speed would be much faster. The reason for this is that in the first form when any control signal (say S2) would change, then every function would be re-evaluated. But, in the second representation, only the D register would be re-evaluated. This effect becomes quite prominent for large modules.

The control signals S1, JMP, S2, S3, S4, ZEROA, TRANA, SHR, ClRL, SUB, ADD and LCID are generated in the control unit which can be initially specified at a design language level, with the user of a flow-table type specification for sequential action. Part of this flow table is indicated in Figure 16. Ultimately, it would be expressed at the gate level, as is done, for example, in Figure 14.

The functional representation for the Arithmetic Unit is given in Figure 19, which consists of two functional modules. If ADD is true, then the contents of A and R are functionally added; while if SUB is true, R is subtracted from A. If ClRL is true, A is circulated left one bit, but if SHR is true, it is shifted right one bit. If TRANA is true, the adder output is transferred into A; whereas if ZEROA is true, A is set to zero.

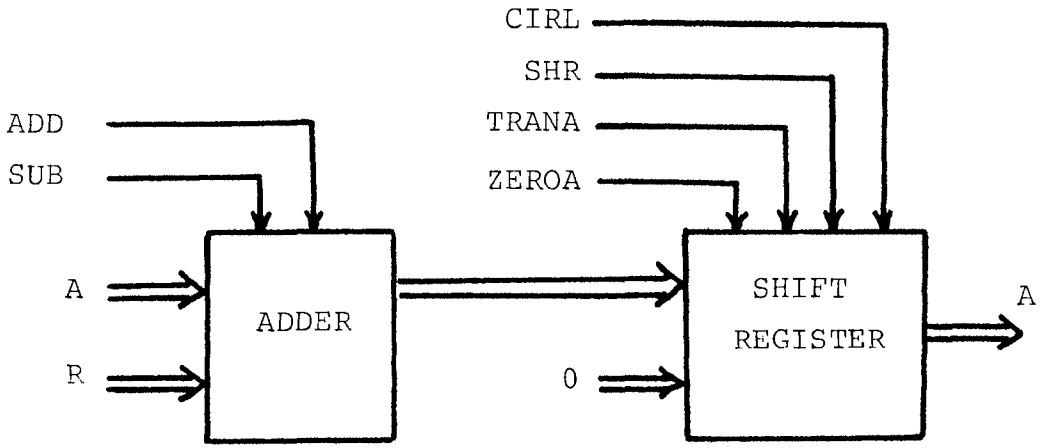


Figure 19: AU in a Standard Module Representation

The memory module is already in a good functional form for this example since standard functional memory modules are available to the system. However, for large memories, interleaved memory modules are just as applicable and effective on a simulation basis as they are on a hardware basis.

From this example, it can be seen that the level of expression and simulation can be varied consistently with the particular simulation task at hand. The level can be varied from the initial design language to more efficient modular representations and to gate level, when necessary.

CHAPTER IV

RESULTS AND CONCLUSIONS

A. Simulation Speed and Storage Analysis

The simulator as described in this dissertation has been implemented in Fortran IV for use on the IBM 360/50. It is presently set to simulate 1000 elements, but this can be changed by changing the dimensions on the common areas of SIMV and STATE and their associated length variables.

The storage requirements for the system are as follows:

Preprocessor	-	90 K bytes
Simulator Root Segment	-	100 K bytes
Mode 1 Simulator Segment	-	50 K bytes
Mode 2 Simulator Segment	-	47 K bytes
Mode 3 Simulator Segment	-	67 K bytes
		354 K bytes

The Mode 1 simulator segment is larger than Mode 2 since Mode 1 contains combinational test generation and Mode 2 and Mode 3 do not.

Additional storage requirements for larger circuits can be computed on the following basis:

Mode 1: 41 * Elements(bytes)

Mode 2: 53 * Elements(bytes)

Mode 3: 65 * Elements(bytes)

These storage calculations are based upon the assumption that: 1) there will be an average of 4 fan-ins and 4 fan-outs for each element, 2) there will be no more than a total of 2 times the number of element changing at any one time.

The time required for simulation is not as easily arrived at as storage. This is because simulation time is extremely problem dependent. Selective trace is one of the causes of this dependency, in that if only a few signals were changing in a circuit with a large number of gates, simulation would appear to be extremely fast. This however, would not be a valid representation of simulation speed. For this reason the simulation times given here will not reflect the selective trace advantages.

An oscillating NAND gate which feeds 24 other AND gates in parallel was timed while oscillating 95 times during Mode 1, Mode 2 and Mode 3 simulation. The total time was 4.7, 10, and 15 seconds respectively. This included a base time of 1.8 seconds which is associated with simulation overhead. Base time is independent of the circuit but directly proportional to

the length of simulation time. This is primarily a consequence of a cyclic TQ table.

The speed of simulation can be determined, in terms of seconds per pass per element per fault using the above information, to be as follows:

1) Mode 1 40 μs
2) Mode 2 100 μs
3) Mode 3 150 μs

When considering the simulation speed of functional simulation, problem dependence plays a much larger part than in gate level simulation. One must not only consider selective trace capabilities, but also the use of data signals,which in actuality is forced selective trace.

In order to determine the type of speed improvements accomplished by functional simulation as well as its dependence upon another factor, consider the following circuit. A very common element used in computer design is that of a register selection circuit. This circuit was simulated at the gate level as an AND-OR register select circuit and then functionally as a register transfer module. This was done for bus lengths of 12 and 24. If one defines the time performance factor (P_t) as the ratio of gate level simulation time to functional level simulation time,then the results of this simulation is as follows:

$$\text{24 bit transfer:} \quad P_t = \frac{167 \text{ sec}}{4.2 \text{ sec}} = 40$$

$$\text{12 bit transfer:} \quad P_t = \frac{84 \text{ sec}}{2.6 \text{ sec}} = 32$$

The simulation time was obtained while making 1100 passes for 32 faults. This is 170μs per pass per fault per transfer for the 24 bit functionally simulated register transfer.

It can be seen from the previous example that the performance factor increases with bus length, as would be expected, since functional elements treat busses collectively.

The storage performance factor for the 24-bit transfer was found to be 12 for the previous example.

B. Conclusion

It is felt that the simulator presented here is truely a system simulator in that it is versatile enough to handle a large majority of the simulation problems associated with the design of computing systems. The range of problem capability can be varied from detailed simulation of asynchronous sequential control circuits to functional or gate level simulation of large modules and their interaction with the total system.

The key of this approach to simulation can be attributed to the increased speed and reduced storage

requirements of functional simulation along with its inherent simplification of system organization and specification.

APPENDIX A

System Implementation

I. System Simulator

The system simulator is organized around a time queuing approach which is used to keep track of the time at which an event occurs in simulation, such as the change in a signal value. The simulator updates the value of each signal at the appropriate time as indicated in the time queue. New entries in the time queue are created as a result of the change in signal value. For example, if the output of an element, which feeds two other elements, has just changed, then, when the output of the first element changes, the two following element outputs must be evaluated and changed at the appropriate time according to the propagation time of each element. Thus, one can see that simulation is a process of following or keeping track of when and what values are changing. Note, that if an element output does not change, it does not effect any other elements and causes no more entries to be made in the Time Queue.

To be able to accomplish this simplified process of simulation, it can be seen that for each element one must know: (1) how to evaluate the outputs of the

elements, (2) what elements they effect (their fan-out), (3) the value of the element outputs and, (4) what signals are input to the element. This information is called the circuit description and is located in the Circuit Description Table (CDT(I,J)). In order to be able to make the index I of the CDT correspond to the element output number (which is used to identify element output signals) the CDT actually contains pointers to the required information such as the fan-out list.

CDT(I,2) is the point to the fan-in list which is located in the Fan-In table (FI). CDT (I,3) contains the pointer to the Fan-Out table (FO) which specifies to where this element output fans out.

CDT(I,1) specifies the element number. This number corresponds to the element type. The element type is the index k of the Function Description Table (FDT) (k,L) which contains parameters used in the evaluation of the element output. These parameters include: (1) the element type number, (2) propagational time delay, (3) the length of the fan-in list plus (4) other parameters for more complicated elements which will be discussed in a later section. The element function type number FDT(k,1) is the number which points to the statement which begins the evaluation procedure for that element type. Transfer to this procedure is accomplished

by a "computed go to" based upon the element function type number.

Note that the evaluation procedure is reached by going from element type to element function type to evaluation procedure. This not only corresponds to the way in which one logically looks at such a classification, but this intermediate level permits a great reduction in the number of evaluation routines required.

The actual value of an element output is contained in the Current Value table (CV(I)). Here the index I is the same index that is used in the CDT. This is so the element output specified by I, in the CDT, has its output value located in the CV table at I.

For the time being, let us consider the Time Queue (TQ) to be an array which contains Element Output Numbers (EON) that are in a state of transition and their Element Output Value (EOV) for the appropriate time at which they change. Time is the index of the Time Queue.

In actuality the TQ does not itself contain the EON and EOV but contains a pointer to the first and last entered EON and EOV which are contained in the Module Transition Table (MTT). This was done to conserve the wasted storage which occurs when one sets aside a certain amount of storage for each time slot and then does not have any entries for some of them. A further

savings in storage is made by making the MTT a chain linked list structure. Chain following need not be done each time an entry is made in the MTT, since the TQ contains the last entered pointer as well as the first entered pointer. Note that some conservation of storage could be obtained by following the chain each time an entry is made.

It is obvious that if one wishes to simulate large intervals of time relative to the largest element delay, one would not want to make the TQ this large since the activity in the TQ is going to be concentrated in the region up to, at most, a distance equal to the largest element delay. For example, if one wishes to simulate ten milliseconds of 10 nanosecond logic this would require a TQ of length 1,000,000 which is not realistic. For this reason, the TQ is a cyclic table of which the length is variable. So, for the example above, the TQ could be made 1000 long and then cycle through it 1000 times. To be capable of entering events which are to occur at large time intervals relative to the maximum propagation delay, the Macro Time Queue (MTQ) is used. The MTQ is itself a Que for the cycles of the TQ. It would contain the EOV, EON and time that they occur relative to the start of the cycle. The index of the MTQ corresponds to the cycle of the TQ to which

these entries apply. The Macro Time Queue Stack (MTQS) is a chained list for the MTQ in the same way as the MMT is for the TQ.

The subroutines which simulate the circuit description as previously described are SIMM1, SIMM2, and SIMM3. SIMM1 is the Mode 1 simulation, SIMM2 is the Mode 2 simulator, and SIMM3 is the Mode 3 simulator. Each of these routines contain in-line element evaluation routines for all gate elements and standard functional elements. Non-standard elements are accessed via subroutine calls. The next section pertains to the actual implementation of these various element types.

II. Element Simulation

In order to evaluate (or simulate) an element, the system simulator performs a "computer go to" to the Element function Type Number (ETN), which is located in the EDT. This ETN is actually the statement number of the first statement in the evaluation procedure. A layout of these ETN will be given later according to Element Type. Thus, to simulate a 2 input AND gate a statement as follows might appear:

```
    1   N = FDT(CDT(FO1,1),3)
        SP(1) = 1
        DO    100    NN=1,N
  100   SP(1)=SP(1).AND. CV(FI(CDT(FO1,2)+NN-1))
```

The number of inputs N is first determined from the FDT. Then an AND operation is performed N times using the values located in the FI list as input values. Note that all 32 bits are ANDED in one instruction, 32 simulations in parallel.

To change an element definition, one need only change this one evaluation procedure. Simulation is hindered in no way by how these elements are evaluated. For this reason a element can represent a gate which is faulty just as easily as another element could represent a fault free gate.

The flexability of this simulator is obtained by providing elements which have a wide variety of capabilities. For example, an element can be larger than just one gate such as a flip-flop or an adder. The function is evaluated at a much higher level for simulation than just evaluating the logic relationship of each gate involved in the module. For example, an adder would be simulative as A=B+C instead of evaluating each logic element in the adder circuit.

In order for the concept of complex elements to be practical, one must consider elements with multiple outputs. Up to this point, element and element output have appeared synonomous. However, now element refers to a collection of outputs. Thus, since the CDT contains element output descriptions then an element with multiple

outputs refer to a group of entries in the CDT. These element outputs of an element are indicated in CDT (I,4) where the first output contains the negative number of the second output, the second contains the positive number of the third, etc. The last contains the number of the first output. This interlinking is used during simulation to determine the fan-out of all outputs whenever the value of the element changes. Each output can have a separate entry in the FDT so that different outputs can have different delay times.

Another interesting (as well as advantageous) characteristic of most functional modules is that a number of the input or output lines can be grouped together and considered collectively as a bus. This permits one to refer only to the bus instead of each line individually. Thus, only one entry in the CDT need be made for a bus instead of one for each line. This same reduction appears when specifying fan-in or fan-out. The only inconsistency appears in where the actual value of each signal going to reside, since there is only one entry in the CV for a value. The solution is in letting the CV be a pointer to the actual values. A similar increase is achieved in speed as was in storage for the cases in which the elements function is merely to select a bus of the input and transfer it to the output. This would correspond to a register

selection module. The increase in speed is possible by transfering the pointer to make the transfer of value, since it is completely compatible with the system simulator.

The pointer points back to the CV and a paging scheme is used to keep track of which locations in the CV are free and which are being used. This information as to page use, length and location is contained in the Bus Value State (BVS).

Module elements can be obtained in several ways. They can exist as standard functional modules, which are very similar to gate elements, and in the simulator subroutine itself for fast access. Modules can also be entered by the user in a variety of forms. These are Fortran, DOL[21] (a computer design language) and gate modules. All of these will be accessed during simulation as subroutines. Fortran will be transferred almost in the same form as user input. A routine called DOLP will translate DOL into executable Fortran code. Gate modules (combinational modules which are specified at the gate level) will also be compiled into Fortran Code in a compiled fashion.

When using functional elements as previously described, one must place more restriction upon how one specifies an element. For example, the inputs of a

simple gate need not be ordered whereas those for an adder element must be. For this reason both inputs and outputs must be ordered for functional elements. The rule for ordering inputs is to start at the top left and number down, assuming all inputs enter the left side of the element. Similarly outputs are numbered going down from the top right side of the element, assuming all outputs leave the right side of the element.

Another convention that must be adopted when dealing with functional elements is the numbering of bits in registers or busses. If the register is numbered from left to right, in order of increasing numerical significance, as indicated in Figure A.1, then this bit pattern would be represented during simulation as indicated in Figure A.2.

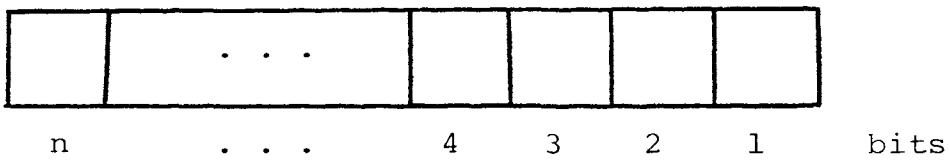


Figure A.1. Bus Representation Convention

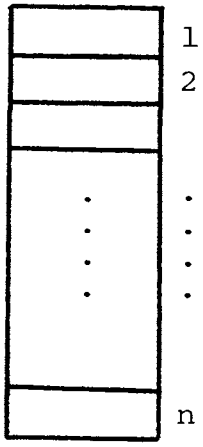


Figure A.2. Simulation Representation of a Bus

It was seen that the best representation for logical variables was to store 32 different machine simulations for the same variable in one word. This, however, is contrary to arithmetic operations in that it would be best to store the total bus or register in one word, so that it can be used directly as an arithmetic variable. A bus of length n used while performing m parallel simulations would be stored during simulation as indicated in Figure A.3. This would then have to be flipped as indicated in Figure A.4 before arithmetic operations could be performed using these values. This is automatically done when using standard functional modules. However, when writing nonstandard Fortran modules, this flip must be performed. This can be done by using the subroutine FLIP$(FRØM,LF,TØ,LT). FRØM is the array being flipped. LF is the length of FRØM in

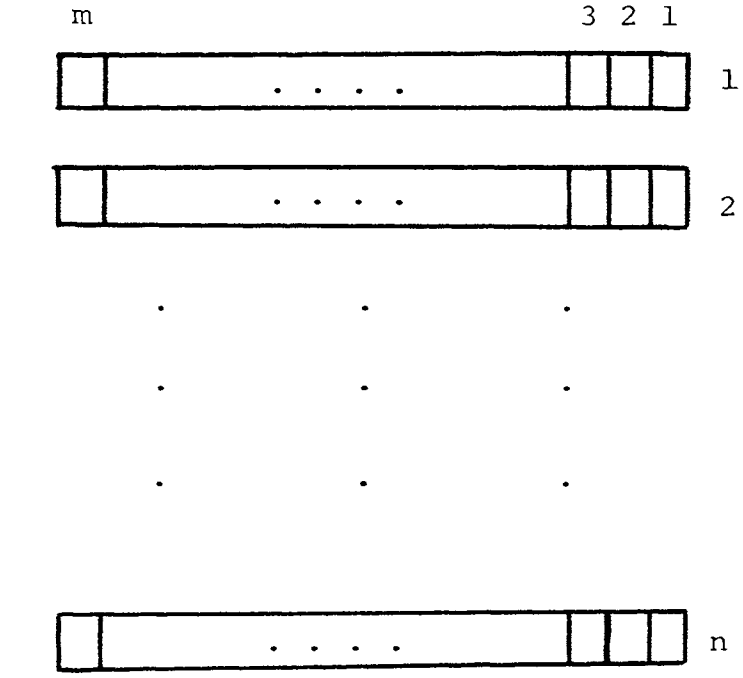


Figure A.3. Logical Bus Representation

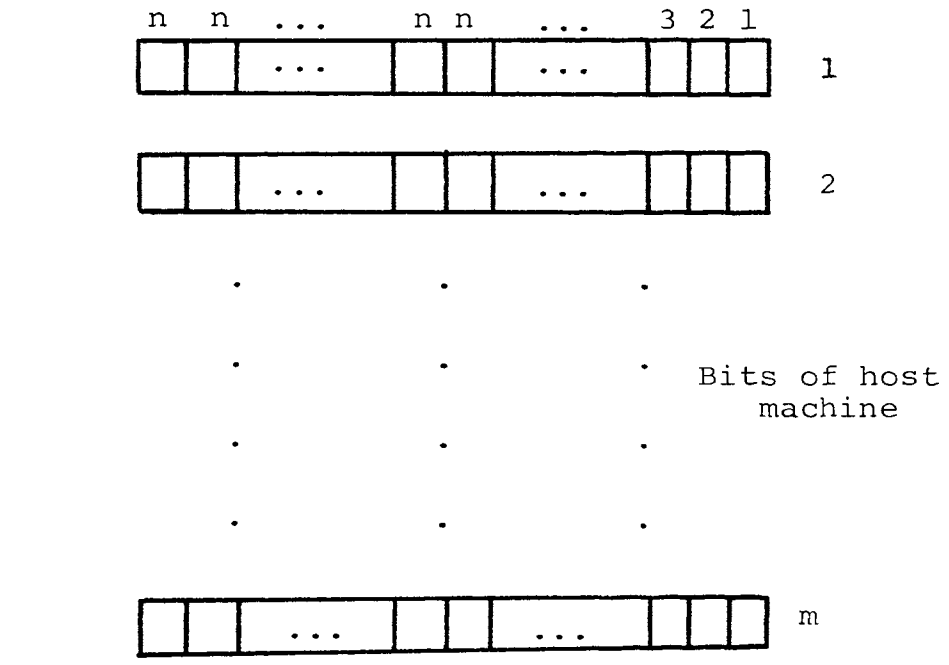


Figure A.4. Arithmetic Bus Representation

words. TØ is the array in which the results of the flip will be placed. LT is the length of TØ in words.

Figure A.3 is referred to as the logical representation while Figure A.4 is referred to as the arithmetic representation.

In the definition of a data signal it was stated that a change in a data signal would not in itself cause a change in the value of the output signal. This is stated more generally by saying that a change in a data signal does not change the state of the circuit. This definition of data signals should only be relied upon after correct timing of the circuit of interest has been verified. The reason for this distinction is illustrated in the example in which the output of an adder module feeds a storage register which is clocked. In actuality the inputs of the adder module should all be control signals since the output will change when any of the inputs do. But since this is immediately buffered by a register this change will not be propagated.

Two other things should be noted about the use of control signals and data signals. First, every element must have at least one control signal. This must be true or the element could never be evaluated. Secondly, a control signal does not always effect the value of an element output other than when it becomes valid.

The logical operations available under Fortran IV G did not include the logical NOT (of all 32 bits) or the logical EXCLUSIVE OR whereas Fortran IV H does provide two such logical functions (LCOMPL, LXOR). To be capable of compiling either under H or G, LCOMPL and LXOR functions are provided when compiling under G but can be removed when compiling under H which is a more efficient compiler.

III. System Generation and Simulation Control

Total machine description (system description) is accomplished by describing the modules of the system. Each module may, for a particular simulation, be a single functional element, hundreds of gate elements or any where between these two extremes. It is the job of INRFAS to set up the tables for simulation as specified by the user for this particular simulation pass. This involves reading in the intermodular system description, reading, and converting the appropriate gate modules and performing signal connection between modules. Since any combination of module descriptions may be requested at a given time, INRFAS must be capable of relocating these descriptions in the simulation structure.

INRFAS is also in charge of generating and manipulating the master and temporary system description which can be used for re-initialization of the simulator from one pass to the next when using the same basic description for each pass. Control for such manipulation is discussed in Appendix B.

Other control instructions are interpreted by INRFAS which are primarly concerned with specifying what type of simulation is to take place. These instructions might specify Mode 1, Mode 2 or Mode 3 simulation, setting up control for combinational test generation or simulation initiation. The actual instructions will be discussed in Appendix B.

Instructions for setting up simulation run control are processed by SCI, SCI2 and SCI3. These instructions pertain to such things as when simulation is to stop, what values are to be printed, when input variables are to change, etc. Due to the different common allocation for Mode 1, Mode 2 and Mode 3 there are three SCI routines, SCI, SCI2, and SCI3, respectively. The appropriate SCI routine is initiated by INRFAS with the *SIMSETUP control card.

RESET, CRDIN, CRDOUT, CRR, READCD and DIS are routines initiated by INRFAS which process the different input and output forms of the system description.

PRINTT and PRINTS are routines used for outputting simulation information during the process of simulation. These appear as special elements to the simulator so that no special testing procedure need be used to determine output requirements.

IV. File Assignment

A number of data files are used in conjunction with TEGAS for the purpose of providing large storage capabilities along with minimizing inter-dependence amoung the various subsystems of TEGAS. Some of the more important files will be described in this section.

File 4 contains the gate description of each gate module that has been specified at the gate level. This is in a numeric form that is generated by the preprocessor from the users mneuomic description.

File 5 is the Module Master File. It contains information about the total system being processed and module information pertinent to the total system.

File 7 is the Fortran Load Module, on which non-standard functional modules are placed until linked to the simulator.

File 11 contains all non-standard Fortran source modules.

File 15 is used as temporary storage for all simulation control instructions.

File 16 contains the master and temporary copy of the system currently being simulated.

File 17 contains the preprocessed system description.

File 18 is a temporary storage file for simulation output.

File 19 is used for temporary data storage during simulation for such things as memory modules.

V. Macro Control and Data Flow

The control and data flow chart for the TEGAS system is given in Figure A.5. Three segments are indicated by the three job step divisions as follows. In Step 1 the preprocessor is executed to do all preprocessing for the job at hand. Step 2 compiles and links any generated non-standard functional modules. In step 3 the simulation of the system is performed.

The preprocessor converts gate modules specified by the user to a description that can be used during simulation. It also processes Fortran and DOL non-standard functional modules and coordinates the calling of these subroutines during simulation. The preprocessor also sets up the system description file as required by simulation.

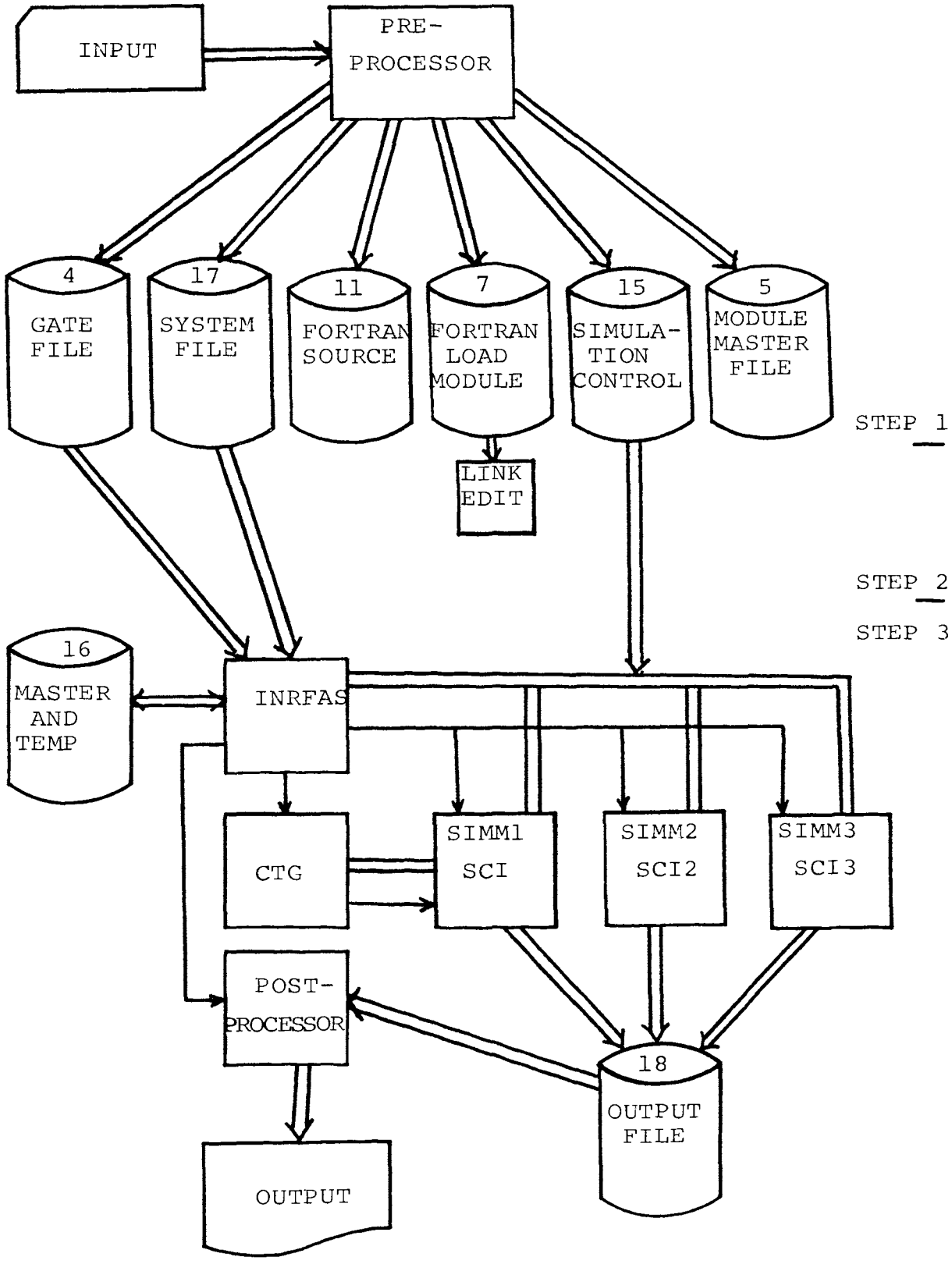


Figure A.5: System Flow Chart

The group of programs labeled CTG is used for combinational test generation. The description of these programs will not be given here since they can be found in reference 15.

At the beginning of step 3 control is passed to INRFAS which reads the instructions sequentially from file 15 and sets up simulation control according to these instructions. As mentioned earlier the task of INRFAS can be divided into two groups, that of organizing the system description for this particular simulation, such as what type of description will be used for which module, and inter-connecting the modules with the system description. INRFAS also must set up the means of controlling simulation while it is being executed. This involves passing control to the correct simulator, to set up control for combination test generation, as well as a number of less important tasks.

The tables used during simulation are generated by INRFAS and are stored in labeled common.

VI. Subroutine Macro Description and Flow

A. SIMM1

SIMM1 performs Mode 1 simulation of systems specified to it by name common tables. The contents of these tables and other important variables are as follows:

```
CDT => CIRCUIT  DESCRIPTION  TABLE

      CDT(*,1)=>  ELEMENT TYPE NUMBER
      CDT(*,2)=>  FAN-IN POINTER   (POINTS  TO  FI)
      CDT(*,3)=>  FAN-OUT POINTER  (POINTS  TO  F0)
      CDT(*,4)=>  MULTIPLE  OUTPUT CHAIN  POINTER
                  CHAIN DETERMINES ORDERING OF  OUTPUTS,
                  FIRST POINTER IS NEGITIVE TO INDICATE
                  BEGINING POINT
      CDT(*,5)=>  FAN-OUT LENGTH

FDT =>  ELEMENT  TYPE  DESCRIPTION  TABLE
      FDT(*,1)=>  ELEMENT FUNCTION   TYPE - POINTER  TO
                  EVALUATION  ROUTINE
      FDT(*,2)=>  ELEMENTS TIME  DELAY
      FDT(*,3)=>  NUMBER  OF INPUTS  TO THE ELEMENT
      FDT(*,4)=>  BUS  LENGTH - 0 OR 1 => SINGLE VALUE

CV(*) => CURRENT  VALUE OF THE SIGNAL

FI(*) => FAN-IN  SPECIFICATION LIST

FO(*) => FAN-OUT SPECIFICATION  LIST

TQ    => TIME QUEUE
    TQ(*,1)  =>  POINTS TO THE FIRST OF A CHAIN OF
                 ENTRIES IN THE MTT

MTT   => SIGNAL  TRANSITION  TABLE
     MTT1(I) => SIGNAL  CHANGING AT  T , TQ(T,1)=I
     MTT2(I) => VALUE   MTT1(I) IS CHANGING TO
     MTT3(*) => CHAIN POINTER  TO OTHER SIGNALS THAT
                ARE CHANGING AT  TIME T

MTQ(*) =>   MACRO  TIME  QUEUE -MTQ'S INDEX IS INCREMENTED
            EVERY  TQ CYCLE IF MTQ=0  NO ENTRY IS MADE IN THE
            TQ  FROM THE  MTQS1, IF MTQ≠0  MTQ IS A POINTER
            TO THE MTQS1 AND MTQS2 WHICH IS ENTERED IN THE TQ

MTQS1 =>    MACRO  TIME  QUEUE  STEP
     MTQS1(*,1)=> CONTAINS THE VALUE CHANGING  AT THIS MACRO-
                  TIME  STEP
     MTQS1(*,2)=>  TIME   OF CHANGE  RELATIVE TO  PIMAX
     MTQS1(*,3)=> CHAIN POINTER  TO  OTHER  ENTRIES IN
                  MTQS1  WHICH ARE CHANGING  AT  THIS
                  MACRO TIME STEP
     MTQS2(*)  => VALUE TO WHICH MTQS1(*,1) IS  CHANGING

BVS  =>      BUS  VALUE STATUS
     BVS(*,1)  => LENGTH OF BUS VALUE  PAGE
     BVS(*,2)  => BEGINING LOCATION  RELITIVE  TO  CV(0)
     BVS(*,3)  => USAGE  OF PAGE

SP  =>  SCRATCH PAD ARRAY-FOR TEMPORARY STORAGE
```

PT	→	PRESENT TIME - POINTS TO TQ
MTS	→	PRESENT MACRO TIME STEP - PØINTS TO MTQ
PIMAX	→	1/2 LENGTH OF TIME QUEUE
LMTTP	→	PØINTER TO NEXT FREE ENTRY IN MTT TABLE
MN	→	ELEMENT WHOSE FAN-OUT IS BEING FOLLOWED - OBTAINED FROM MTT1 AT PT
FØ1	→	FAN-OUT OF MN
TQPT	→	POINTER TO MTT ENTRY THAT IS CURRENTLY BEING PROCESSED
INIT	→	IF (INIT=1) SELECTIVE TRACE IS NOT USED IF (INIT=0) SELECTIVE TRACE IS USED
MTTPMA	→	LENGTH OF MTT
TT	→	PROPAGATION TIME OF ELEMENT FØ1

The flow chart for SIMM1 is given in Figure A.6 and an explanation of each block or decision point is given as follows:

Search TQ for next PT which contains activity.

Has the value of MTT1(PT) just changed and is INIT=0?

Is this a bus line?

Has bus value changed?

Free old bus page.

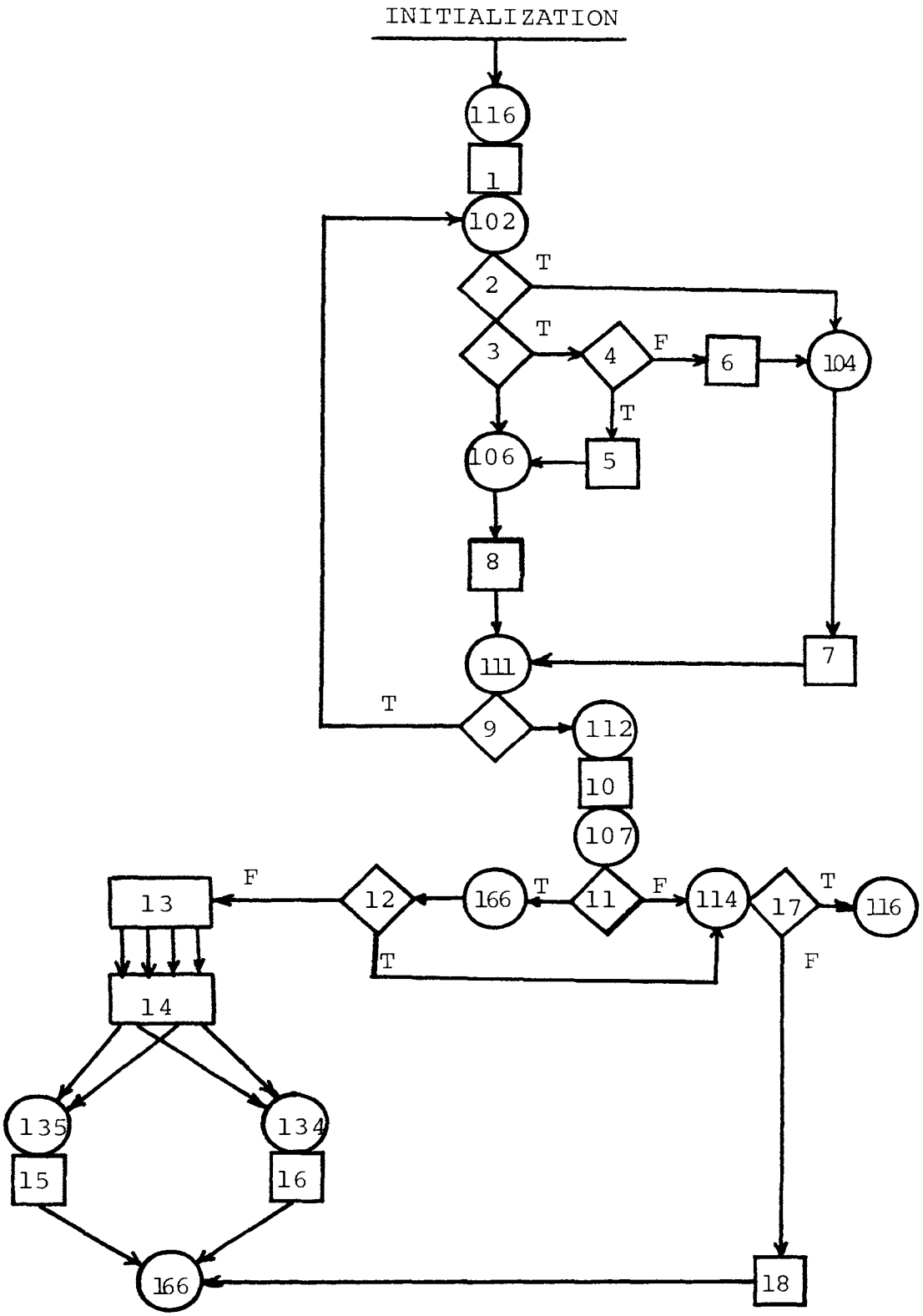


Figure A.6: SIMM1 Flow Chart

Free new bus page.

Enter 0 in MTT1 to indicate that this signal has not changed.

Update value of signal.

Is there any more entries in MTT for PT?

Reset TQPT to first entry in MTT for PT-MN.

Did MN just change from its previous value?

Has all fan-outs of MN been followed?

Determine element function type and do a computer "GØ TØ" to that value.

Evaluate the output value(s) of FØ1 and place it (them) in SP.

Elements with single outputs return to 135 to have its output value entered in the MTT at the appropriate time, at TP+TT in the TQ.

Elements with multiple outputs return to 134 to have its outputs entered in the MTT at the appropriate time,at TP+TT in the TQ.

Is this the last element changing at PT?

Determine next element changing and determine its fan-out length.

B. Subroutine SIMM2

Subroutine SIMM2 is just like SIMM1 except in the following cases:

1. Common/SIMVM2/ is added to provide storage for the indeterminate value.

2. In Block 2 of SIMM1's flow chart the indeterminate bit must be checked for changes.

3. The evaluation routines in Block 14 must be changed to include an indeterminate bit.

4. In Block 15 and 16 the calculated indeterminate value must also be placed in the MTT.

C. Subroutine SIMM3

Subroutine SIMM3 is just like SIMM1 except in the following cases:

1. Common /SIMVM2/ and common /SIMUM3/ just be added to provide storage for the Potential Error (PE) value and the New Value (NV), respectively.

2. In Block 2 of SIMM1's flow chart the PE must also be checked for changes in value.

3. The evaluation routine in Block 14 must be changed to include PE and NV.

4. In Block 15 and 16 the calculated PE and NV must be placed in the MTT. Changes in CV produce entries at PT+TT where changes in NV produce changes at PT+TT+AT where AT is the ambiguity in time delay. Changes in NV are distinguished from changes in CV by using negative MN's for changing NV's. Thus, it appears as if two simultaneous simulations are occuring in parallel. These are the simulation of changes in CV (positive MN's) and simulation of changes in NV (negative MN's).

D. Subroutine INRFAS

Subroutine INRFAS processes system description and simulation control cards. The definition of and format for these control cards can be found in Appendix B. INRFAS has access to the same common area as SIMM1 and the variables used here indicate the same quantities as specified in SIMM1. Other important variables are as follows:

NXTFØ → Next available entry in the FØ table

NXTFI → Next available entry in the FI table

NXTFDT → Next available entry in the FDT table

NXTCDT → Next available entry in the CDT table

MODE → Indicates which mode of simulation is being performed

LCDT → Length of the CDT table

LFØ → Length of the FØ table

LFI → Length of the FI table

LMTQ → Length of the MTQ table

LNMTQS → Length of the MTQS tables

LBVL → Length of the BVS tables

LTQ → Length of the TQ table

LCVB → Length of the CVB table

LMASK → Length of the MASK common block

LFTA → Length of the FTA common block

LFAULT → Length of the FAULT common block

The flow chart for INRFAS is given in Figure A.7.

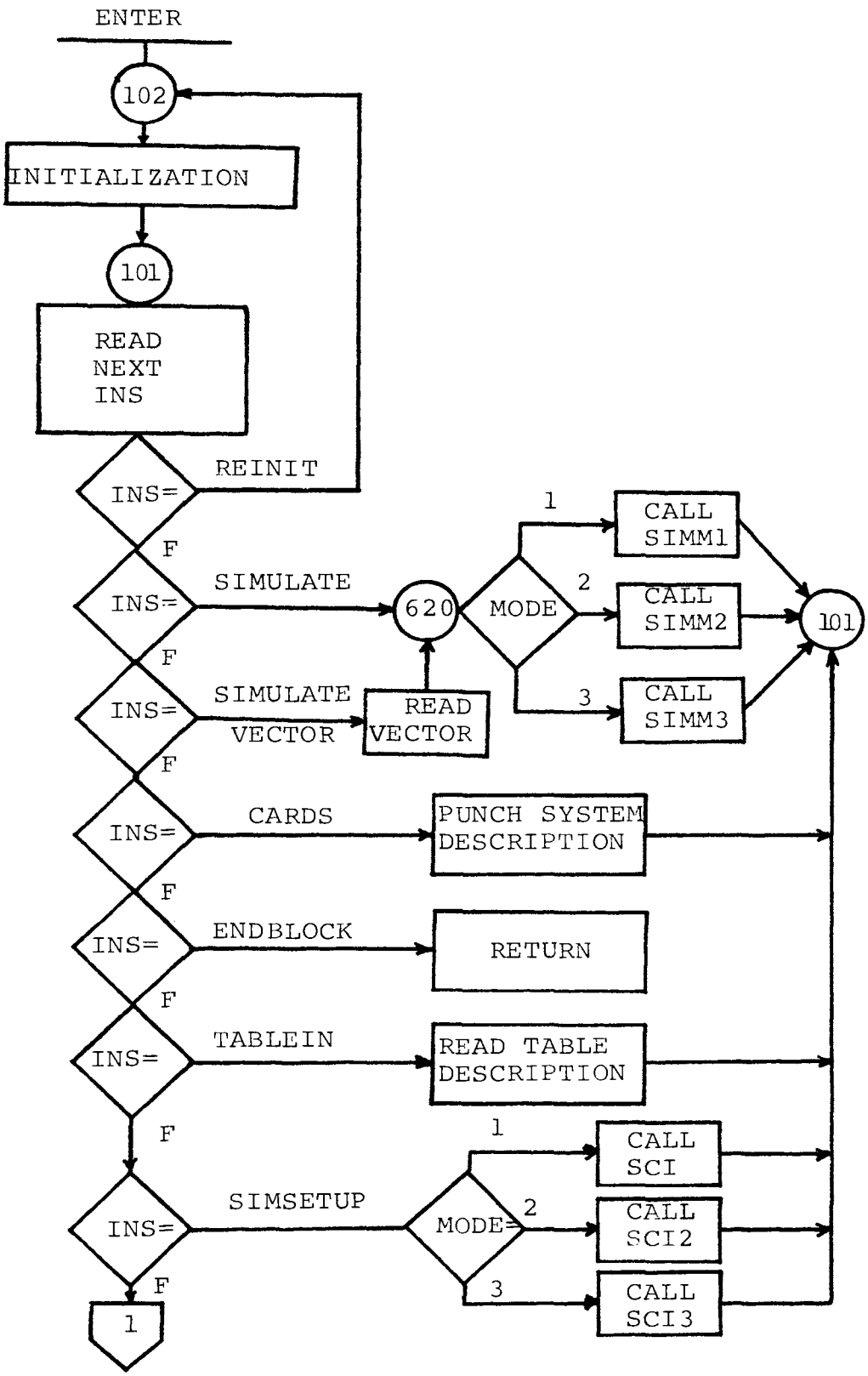


Figure A.7. INRFAS Flow Chart

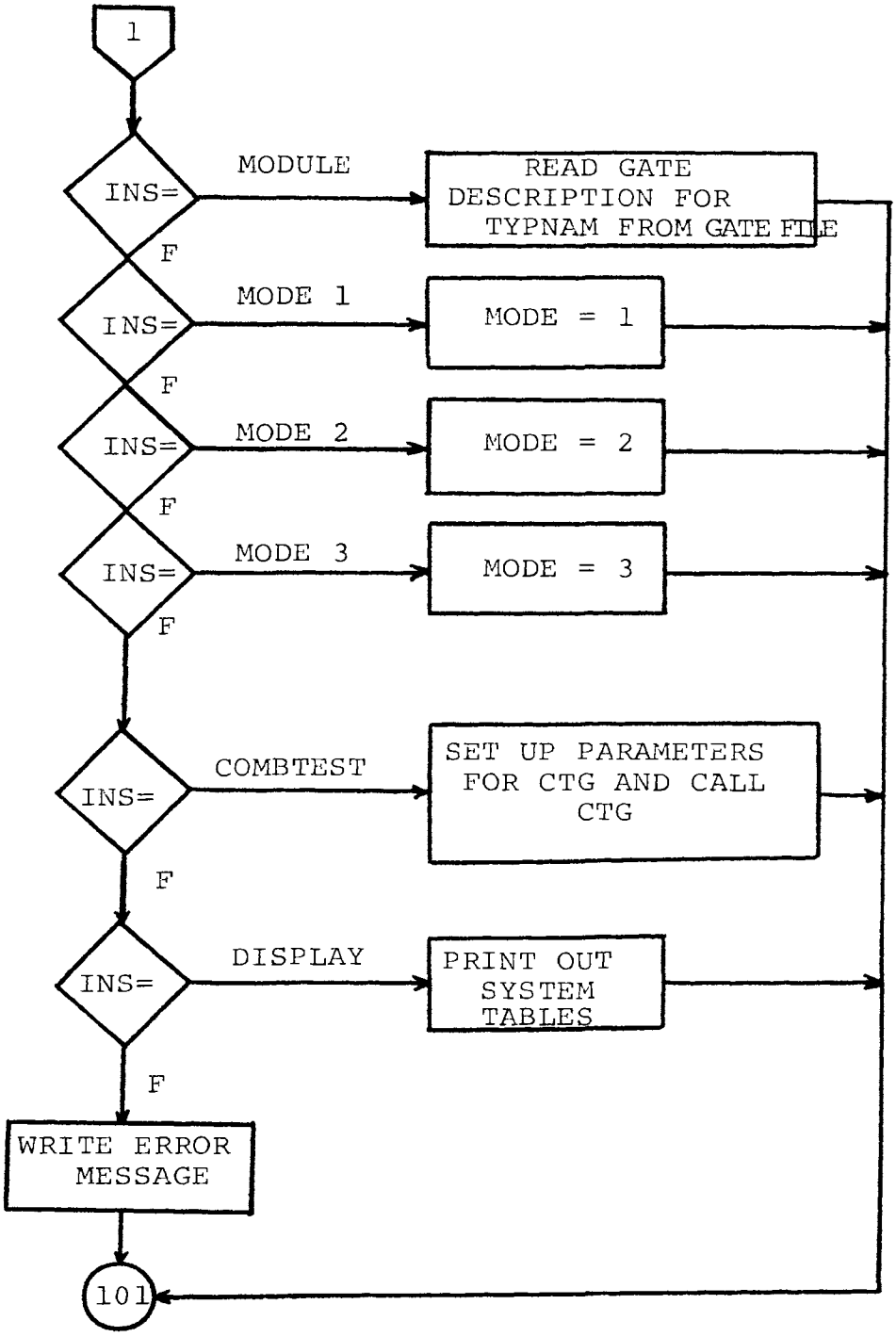


Figure A.7 (continued): INRFAS Flow Chart

Appendix B

User Instructions

Appendix B is concerned with the instructions that are used to describe and simulate a digital system.

A system is described in terms of the modules of that system. Thus, the system description is concerned with the interconnection of modules. A module in general can be one element or a number of elements. If a module is composed of one element, the module is completely described by describing that element. However, if a module consists of a number of elements, the module must be described in terms of the elements that are used to form it. From this it can be seen that to describe a system one must:

a) describe the interconnection of modules (the system description).

b) describe the modules in terms of the elements of the module (the module description).

c) describe the elements used in both the system description and the module description (the element description).

These three aspects of describing a system along with simulation control will be the topic of this appendix. Due to the inability to preprocess description

of modules or systems which contain functional elements it will be assumed here that: a) module descriptions contain only gate elements, b) and that the system description will be described directly to the simulator.

I. Module Specification

Processing of a module is initiated by issuing the following control card:

*MØDULNAM module name

where the * is in column 6, MODULNAM is in columns 7-14 and the actual 8 character module name is in columns 16-23. The module name can be placed anywhere in this field but blanks are not valid characters in the module name since all blanks are squeezed to the right.

Following the MØDULNAM card the elements used in this module description are given. The element description constitutes a block of information. It must start with the following control card:

*G.TYPES

and end with the following control card

*ENDBLØCK

the card following *G.TYPE is:

CTPMAX FINMAX FORMAX TEST

according to the format 3I8, 2A4.GTPMAX is the number of gate types that are going to be described for this module.

FINMAX is the maximum fan-in that any of these gate types can have. F0TMAX is the maximum fan-out that any of these gate types can have.

Note that GTPMAX, FINMAX and FOTMAX must be right justified.

TEST is the module name with which these gate types are to be associated.

Between the previous card and the endblock card there exists one card for each gate type. This card is as follows:

USRGTP NGATYP L0GTYP FINLIM F0TLIM NOMDEL MINDEL MAXDEL

according to the format 2A4, 7I8.

USRGTP is the gate type name that will be used in the circuit description to denote a gate as being of this type. These must appear in alphabetic order.

NGATYP is a numerical gate type which can be used as a numerical sequence number.

L0GTYP is the logical type (or element function type). These will be given along with the standard functional modules in the section on element specification.

FINLIM is the number of inputs of this gate type.

FOTLIM is the maximum fan-outs, the number of gates this gate type can drive.

NOMDEL is the nominal propagational delay time of this gate type.

MINDEL is the minimum propagational delay time of this gate type.

MAXDEL is the maximum propagational delay time of this gate type.

MINDEL and MAXDEL need be specified only when mode 3 simulation is to be performed.

After the G.TYPE block is the CIRCUIT block. This block specifies, in mnemonic form: 1) the logical type, 2) fan-in, and 3) fan-out of each gate.

The first card of this block is:

*CIRCUITS

where the * is in column 6 and CIRCUITS is in columns 7-14.

This block must also end with an endblock card.

Cards between the CIRCUITS card and ENDBLOCK card can be one of two formats. The second format is for continuation of the first. The first format is as follows:

Column	Contents
1-5	Not recognized
6	Blank
7-18	12-character name of gate being described
19,20	Blank
21-28	8 character gate type name
29	List type
30-41	12-character gate name
42	List type
43-54	12-character gate name
55	List type
56-67	12-character gate name
68	List type
69-80	12-character gate name

The format for the continuation card is as follows:

Column	Contents
1-5	Not recognized
6	C
7-15	Not recognized
16	List type
17-28	12-character gate name
29-80	Same as preceeding card

The list type field can be either 0, Ø, 1 or I. A 0 or Ø indicates that the following gate name is to be entered in the fan-out list of this gate. A 1 or I indicates that the following gate name is to be placed in the fan-in list of this gate.

List type characters need not be repeated if they are of the same type as the last list type character for this gate.

Gate names must be unique within a module and multiple specifications with the same gate name will be concatenated.

II. Element Specification

In the previous section it was shown how to specify element types that would be used in a module description. Element types must also be specified in conjunction with the system simulator. This is done by placing in the FDT table, in the next available location, a) the element function type, b) time delay, c) number of inputs and, d) bus length, respectively.

The element functional types available at present are listed as follows:

1. N-INPUT AND GATE
2. N-INPUT OR GATE
3. INVERTER

4. PRIMARY INPUT
5. PRIMARY OUTPUT
6. N-INPUT NAND GATE
7. N-INPUT NOR GATE
8. N-INPUT PRIMARY INPUT
9. N-INPUT PRIMARY OUTPUT
10.
11. DELAY ELEMENT
12. N-INPUT XOR GATE
13. N-2-INPUT AND-NOR SELECT
14. N-4-INPUT AND-NOR SELECT
15. J-K FLIP-FLOP
16. S-R FLIP-FLOP
17. S-R-T FLIP-FLOP
18. T-FLIP-FLOP
19. D-FLIP-FLOP
20. FAULTY N-INPUT AND GATE
21. FAULTY N-INPUT OR GATE
22. FAULTY N-INPUT NAND GATE
23. FAULTY N-INPUT NOR GATE
24. FAULTY N-INPUT XOR GATE
25. FAULTY INVERTER
26. BUS AND MODULE
27. BUS OR MODULE
28. BUS NAND MODULE
29. BUS NOR MODULE

30. REGISTER TRANSFER ELEMENT

31. BUS SELECTION ELEMENT

32. REGISTER CONCATENATION ELEMENT

33. SUBREGISTER ELEMENT

34. ADDER ELEMENT

35. SHIFTER ELEMENT

36. DECODER ELEMENT

37. MEMORY ELEMENT

38. BUS INVERTER

39.

40-45. 5-INPUT NON-STANDARD FUNCTIONAL ELEMENT

46-50. 10-INPUT NON-STANDARD FUNCTIONAL ELEMENT

51. STOP SIMULATION

52. MACRO-TIME QUE UPDATE

53. CLOCK

54. PRINT VALUES

55. PRINT REAL TIME

56-60.

60-100.

The register transfer module is indicated in Figure B.1. AS through NS are control signals which when equal to 1, place the data bus value of A through N, respectively, on the output bus. The output value does not change as a result of A changing if AS=0. If more than one control value equals 1 at the

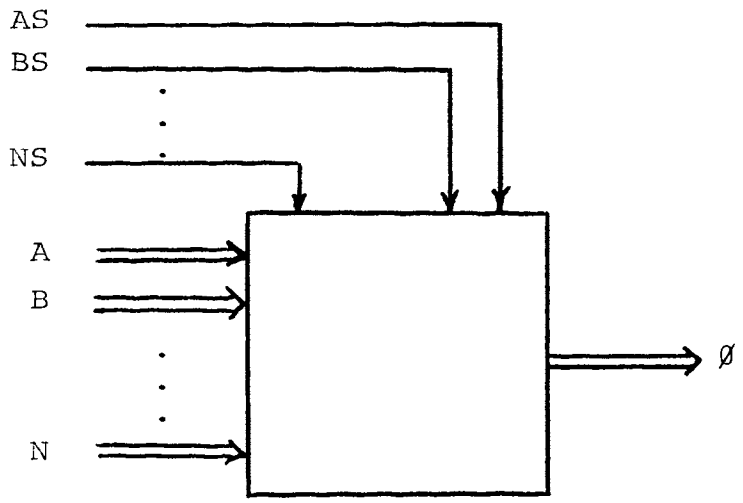


Figure B.1. Register Transfer Module

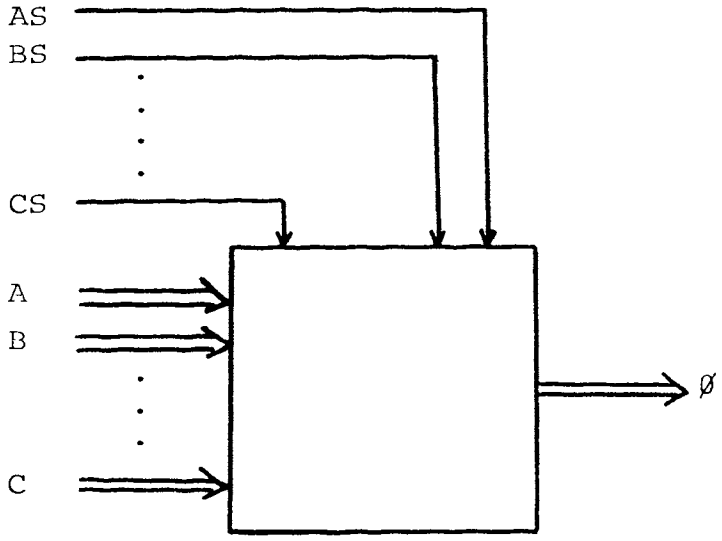


Figure B.2. Bus Selection Module

the same instance, the output is the logical OR of the respective data bus lines.

The bus selection module (Fig. B.2) is the same as the register transfer module except that it does not contain memory for storing the output when the control signals go to zero.

The register concatenation module of Figure B.3 is used to concatenate two busses. Busses A1 through AN are concatenated in their respective order to form the bus output 0. Here A1 through AN must be control busses unless a control line (not shown) is used to initiate proper evaluation of the module.

The subregister module of Fig. B.4 is used to split a register into two or more busses or lines. The bit ordering from input to output remains the same. But the number of bits per bus is smaller for the output busses than for the input bus. Here also A is a control bus unless an additional control line is introduced to initiate evaluation.

The adder module is indicated in Figure B.5. A and B are the busses that are to be added if AD=1. B is subtracted from A if S=1. The result is placed in SUM. If an overflow condition is detected OF is set to 1. Whether A and B are data busses or control busses will depend upon characteristics of the adder being simulated.

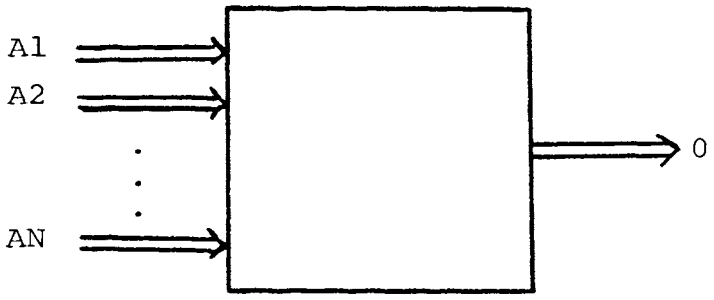


Figure B.3. Register Concatenation Module

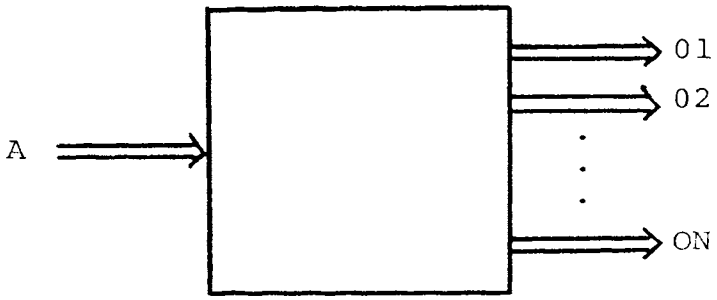


Figure B.4. Subregister Module

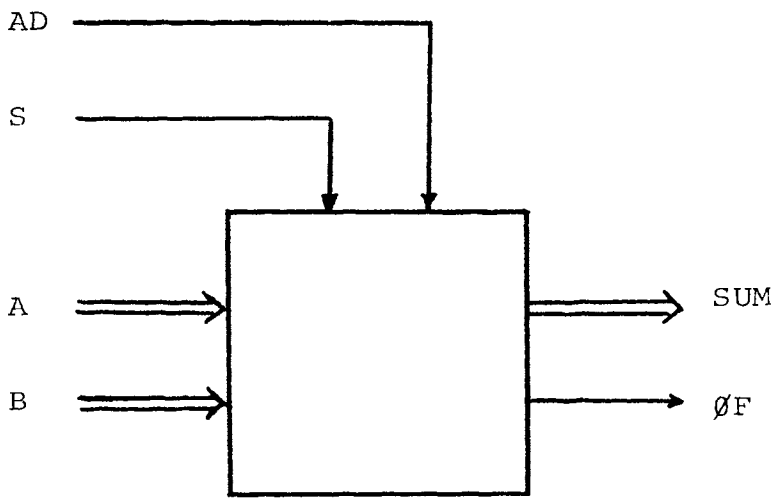


Figure B.5. Adder Module

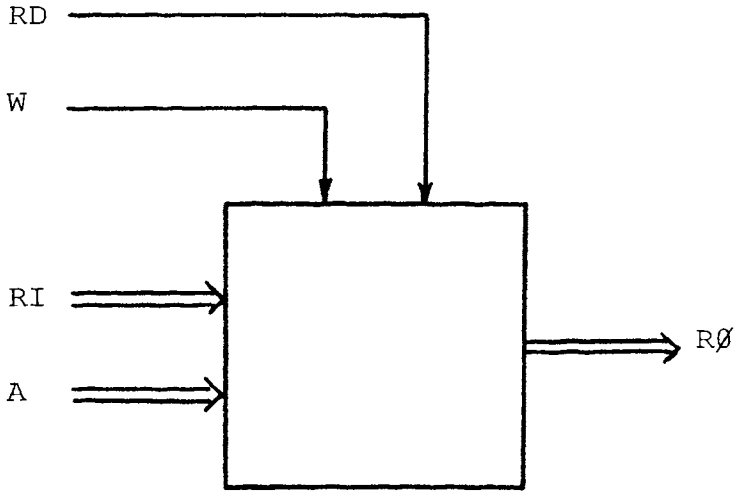


Figure B.6. Memory Module

A memory module is indicated in Figure B.6. If RD is 1 the contents of location A is placed in RØ. If W=1, RI is placed in location A. Simulated memory words are placed on disk as an intermediate means of storage. Pages of simulated memory are brought in as requested. For this reason successive words should be accessed when at all possible so that a minimum of page swaps will be required.

A shift module also exists as a standard functional module. Indicated in Figure B.7. is such a module. Here if SHIFT=1, A1 is shifted N places and placed in A, if CIRC=1, A1 is circulated N places and placed in A. If RIGHT=1, the shift or circulate is right whereas if RIGHT=0, the shift or circulate is left. Note that N is an array which contains the binary number of bits to be shifted or circulated. Thus, the number of bits being shifted might be different from one machine being simulated to the next. It should be noted that for simple shift or circulate operations a gate level implementation might prove to be the better approach.

A standard functional decoder module is shown in Figure B.8 which is available by specifying the module as having a functional element type of decoder module (36 as indicated previously). The location of the array K specified by the numerical value of I is set to one. All others are set to zero. The first element

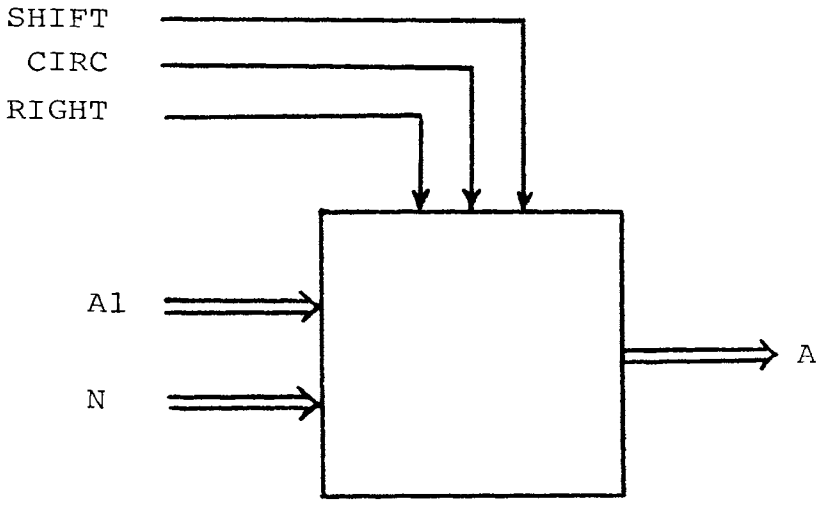


Figure B.7. Shift Module

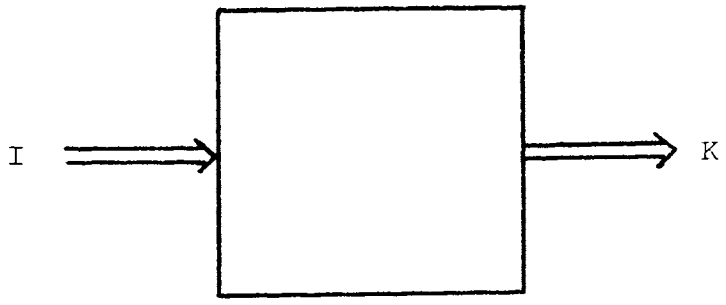


Figure B.8. Decoder Module

of K is numerically element zero. The decoder module can have either I as a control array or another variable (not shown in Figure) as the control variable, depending upon the module being simulated.

Non-standard functional elements can also be created. These can be of two forms: a) Fortran or, b) DOL, a computer design language.

The input format for a Fortran element would be as follows:

```
C     *FØRTRAN:      EFTNO
C     *INPUTS:       X,Y, B(24), A(24)
C     *ØUTPUTS:      Z, D(24)
C     *INTERSTA:     Q,S

                  FORTRAN SOURCE

C     *ENDBLØCK
```

Here *FØRTRAN indicates that the following block contains a Fortran element description whose element function type number will be EFTNO. The EFTNO has to be one of the non-standard functional elements and the number of inputs specified in *INPUTS: must be less than or equal to that indicated by the non-standard functional type.

The non-blank list following *INPUTS:, *ØUTPUTS, and *INTERSTA:specifies the number and order of primary inputs, outputs and internal states, respectively. This must be consistent with the number and order as specified in the system simulator.

*INTERSTA can be used to insure the retention of internal variable values between execution if an overlaying process is used in functional element simulation.

The input format for a DOL element is very similar to the input format for the Fortran element.

```
C     *DØLEMT:      EFTNØ
C     *INPUTS:      A,Z,Q(24),N
C     *ØUTPUTS:     X,ZB(24)
C     *INTERSTA:    W,V

                  DOL SOURCE

C     *ENDBLOCK
```

Here *DØLELMT: indicates that the following block is an element described by DØL.

III. System Description Specification

To indicate the start of the system description and simulation control specification phase, one must issue the following control card:

*UØPTIØNS

The end of this block must be indicated by the end block control card which is as follows:

*ENDBLØCK

The appropriate mode of simulation can be selected by issuing one of the following control cards:

MØDE 1

MØDE 2

MØDE 3

which selects Mode 1, Mode 2 or Mode 3 simulation respectively. If no MODE card is used simulation defaults to MODE 1.

The system description can now be read in table form. One control card is required to do this. Which is:

TABLEIN

The simulation tables can then be read in by name with the following card:

NAME START END

This card will cause the table of name NAMW to be read by subroutines READCD, RDCD2 and RDCD3 starting at START and ending at END. The recognized table names are as follows:

Table	Format
CDT	15I5
FI	16I5
FØ	16I5
FDT	16I5
MTT	12I5
BVS	12I5
MTQS	16I4

For all other tables, START indicates the number of cards to follow. Each card contains one entry for that particular table.

START and END are numeric quantities which are right justified to columns 20 and 25 respectively. These are read under a 2I5 Format. The first is the subscript value and the second is the value that is to be placed in the named table at the position specified by the first number. The tables that are read in this way are CV, MTQ and TQ. All tables are initially set to zero so that only non-zero entries need be made in these tables.

Simulation control elements exist as elements 1-10. For this reason all tables must be numbered starting at 11.

One must terminate this block of table specifications with the following control card:

ENDBLØCK

Note that there is no * in column 6.

At this point simulation control pertinent to the system description can be set up. This block of information is begun with the following control card:

SIMSETUP

It is ended with an end block control card. The actual instructions used for simulation control set up will be described in the next section of this appendix.

Since the system description is in terms of the highest level of simulation then in order to simulate a module at a lower level, the module to be so simulated must be indicated. This is done with the following control card:

MODULE NAME

where NAME is the 8 character module name of the module to be described at the gate level starting in column 17. Following this, another SIMSETUP block can be used to set up simulation control pertinent to this module.

One MODULE card should be used for each module that is to be simulated at the gate level.

After the system has been completely constructed as required for the particular simulation task at hand, then simulation can be initiated with the use of the SIMULATE control card.

Other control cards are available in order to provide more versatility to the simulator. These are as follows:

REINIT which reinitializes the simulator so that a new system or configuration can be constructed.

CARDS produces a card copy of the constructed system in a form which can be read in at a later time.

DISPLAY is a command to display the present system being constructed. This display is in a table form

COMBTEST is a series of control cards which permit different modes of combinational test generation to be performed.

SIMULATE VECTØR is a qualified control card which indicates that an input vector is to be read in before simulation is to be initiated. This is a default type control card to be used in place of a SIMSETUP and SIMULATE. The vector is a binary vector starting in column 1 and ordered according to the ordering of primary inputs to the simulator.

If only one module is to be simulated and it is to be simulated at the gate level, then one need not specify a system description. The entire module then becomes the system.

IV. Simulation Control Specification

Simulation control is the set of instructions which determine how the system just specified is to be simulated. This involves specifying, 1) the initial state of the machine, 2) at what time certain inputs are to change, 3) what information is to be printed, and 4) how simulation is to stop.

As with all blocks of information, this block begins with a special control card, SIMSETUP, and ends with an ENDBLOCK control card.

Following will be an explanation of the instructions that are available for use at this time for simulation control. This, however, is not intended to be a complete set of the best approach for input and output information handling. But this is rather a more easily understood method of controlling simulation problems that are simple in nature.

All instructions obey the following format unless otherwise stated:

Column	Field Name	Contents
1-6	A	blank
7-14	B	left justified instruction word
17-28	C	left justified signal name
30-35	D	clarification instruction field
37-48	E	right justified numeric field
51-54	F	context clarification field
60-71	G	right justified numeric field

SET signal name TO DEC signal value

SET signal name TO BIN

binary signal value

The SET instruction sets the initial value of the signal, specified by the C field, to the value specified in the E field if it is a decimal value. If a binary value is used this value is specified in the field as specified above. The D field specified a decimal value when TO DEC is used or a binary value when TO BIN is used.

STØPSIM AT time

STØPSIM sets control to stop simulation at the time specified. This is optional since simulation is automatically stopped at the end of the MTQ. The time is an integer and is specified in the same units as used for element propagation delay. Time is indicated in the E field

CHANGE signal name TØ DEC signal value AT time

CHANGE signal name TØ BIN AT time

binary signal value

The CHANGE instruction causes the signal indicated in the C field to change to the specified value at the time indicated in the G field. The TØ DEC and TØ BIN have the same meaning as in the set instruction.

INIT TØ time

The INIT instruction causes the selective trace feature to be turned off until the time indicated. This can be used to insure proper initializations.

REALTIME AT time

The REALTIME instruction causes the time of day to be printed at the simulation time specified.

```
PRINT    signal name    AT    time
PRINT    signal name    AT    time     REPT       time
PRINT                   AT    time
 list
PRINT                   AT    time     REPT       time
 list
```

The PRINT instruction causes the signal name or list of signal names to be printed at the time indicated in the E field. If a REPT is specified, then this print will be repeated at intervals specified by the G field starting at the time in the E field.

The list specifies a group of signal names whose values are to be printed. The format for this list is as follows:

```
    NUM
 signal name    signal name
```

NUM is a right justified number to column 10 which gives the number of signal names to be read. The signal names are left justified, 12 character, maximum length, names starting in columns 7, 20, 33, 46, 59.

A default print is also used in conjunction with the default stop simulation. This automatically prints all primary outputs just before simulation is stopped.

Automatic fault insertion can be accomplished by using the GENERATE instruction which is as follows:

GENERATE	ALL	FAULTS	
or			
GENERATE	fault type	FAULT	ØN
GATE	gate name	LEAD	leads signal name
		LEAD	leads signal name

The first instruction generates and simulates all faults after fault collapsing. The second instruction generates and simulates faults of the type specified in the C field. Fault types are either SA1 or SA0. The card following specifies what gate this fault is on and what lead of that gate it is on. The next card indicates that a fault is to exist on the lead specified by the E field. Here the E field is used as a name field. Any number of GATE cards may appear after a GENERATE card and any number of LEAD cards may appear after a GATE card.

V. Sample Simulation

To illustrate the use of some of the simulation features, the control cards for simulation of the circuit cards for simulation of the circuit given in Figure 6 are given as follows:

```
      *MODULNAM RACE
      *G.TYPES
        7       10       10RACE
AND42            1        1        2       10        4
AND43            2        1        3       10        4
INV11            3        3        1       10        2
OR42             4        2        2       10        4
OR63             5        2        3       10        6
PI               6        4        1       10        1
PO               7        5        1       10        1
      *ENDBLOCK
      *CIRCUITS
       X1           PI        0  G              B
      C           D
       X2            PI       0   A              H
      C            B             C
       G             INV11    I  X1          0  A
       A             AND42    I   G              X2
      C          0 Y  1
       H             AND42    I  X2             Y1
      C          0  Y1
       C             INV11    I  X2          0   E
       D             INV11    I  X1          0   E
      C           F
       B             AND42    I  X1             X2
      C          0 Y2
       E             AND42    I  C              D
      C          0 Y2
       F             AND43    I  D              Y1
      C            Y2        0 Y2
       Y2            OR63     I  B              E
      C            F         0   F
       Y1            OR42     I  A              H
      C          0 H             F
      *ENDBLOCK
      *UOPTIONS
       MODULE   RACE
```

```
 DISPLAY
 SIMSETUP
 SET       X1              TO DEC          -1
 SET       X2              TO DEC          -1
 SET       Y1              TO DEC           0
 SET       Y2              TO DEC          -1
 SET       B               TO DEC          -1
 CHANGE    X1              TO DEC           0  AT     2
 PRINT                     AT               1 REPT    2
    4
 X1           Y1              F         Y2
 STOPSIM                   AT             40
 ENDBLOCK
 SIMULATE
 ENDBLOCK
*ENDBLOCK
```

The major block and their sequence should also be noted.

VI. BIBLIOGRAPHY

1. E.G. Manning and H.Y. Chang, "A Comparison of Fault Simulation Methods for Digital Systems", Digest of the First Annual IEEE Computer Conference, pp. 10-14, Sept. 1967.

2. R.W. Downing, J.S. Powak and L.S. Tromenaksa, "No. 1 ESS Maintenance Plan," BSTJ Vol. 43, pp. 1961-2020, September 1964.

3. S. Seshu and D.N. Freeman, "The Diagnosis of Asynchronous Sequential Switching Systems", IRE Trans. on Elec. Comp., Vol. EC-11, No. 4, pp. 459-465, August, 1962.

4. F. Hardie and R. Suhocki, "Design and Use of Fault Simulation for Saturn Computer Design", Western Electronic Convention and Show, August 23-26, 1966.

5. R. Marsh, "A General Logic Simulation Program", Software Age, November, 1969, Vol. 3, No. 11, pp. 28-36.

6. P.R. Menon, "A Simulation Program for Logic Networks", Bell Telephone Labs Technical Report, 1965, MM 65-1271-3.

7. G. Smith,Personal Communications at Mid-West Probability Workshop, Lake of the Ozarks, Missouri.

8. D.B. Armstrong,Personal Communications at Mid-West Reliability Workshop, Lake of the Ozarks, Missouri.

9. S.A. Szygenda, "Sequential Analyzer Program Capabilities and Operating Procedure", BTL Report, MF 14-670831.1, August 1967.

10. S.A. Szygenda, "Digital Simulation and Analysis of Gates", BTL Report, MF 14-680215.1, February, 1968.

11. S.A. Szygenda, "A Software Diagnostic System for Test Generation and Simulation of Large Digital Systems", Proceedings of the National Electronics Annual Conference, December, 1969.

12. S.A. Szygenda, "TEGAS - A Diagnostic Test Generation and Simulation System for Digital Computers", Proceedings of the Third Hawaii International Conference on System Sciences, January 1970.

13. D.M. Rouse and S.A. Szygenda, "Models for Functional Simulation of Large Digital Systems", Digest Record of the Joint Conference on Mathematical and Computer Aided Design, October, 1969.

14. S.A. Szygenda, D.M. Rouse and E.W. Thompson, "A Model and Implementation of a Universal Time Delay Simulator for Large Digital Nets", AFIPS Proceedings of the Spring Joint Computer Conference, May 1970

15. S.A. Szygenda and G. Goldbogen, "Implementation and Extensions of Multi-Dimentional Path Sensitizing in a Simulation and Diagnosis System", Proceedings of the 7th Annual Allerton Conference on Circuit and System Theory, October 1969.

16. S.A. Szygenda and R.J. Smith, II, "A Preprocessor for Software Diagnosis and Simulation of Digital Systems", Proceedings of the Second Annual Houston Conference on Circuits, Systems and Computers, April 20-21, 1970.

17. E. Manning and H. Chang, "Functional Techniques for Efficient Digital Fault Simulation", Digest of the First Annual IEEE Computer Group Conference, September 1967.

18. L.C. Bening Jr., "Accurate Simulation of High Speed Computer Logic", IEEE Repository.

19. D.L. Smith, "Models and Data Structure for Digital Logic Simulation", Masters Thesis Massachusetts Institute of Technology, 1966.

20. S. Seshu, "The Logic Organizer and Diagnosis Programs", Report R-226 Coordinated Science Laboratory,University of Illinois, Urbana, Illinois (AD605927) 1964.

21. D.M. Rouse, "A Design Oriented Computer Design Language", Masters Thesis, University of Missouri-Rolla, Rolla, Missouri, 1969.

VII. VITA

David Michael Rouse was born on September 16, 1945 in Joplin, Missouri. He received a Bachelor of Science degree in Electrical Engineering from the University of Missouri-Rolla in June, 1967. He has been enrolled in graduate school at the University of Missouri-Rolla since September, 1967. At this time he married Jeanne Martin who majored in Elementary Education.

He was on the staff of the Electrical Engineering Department from September, 1967 to June, 1969. He received his Master of Science degree in Electrical Engineering in June, 1969.

During the summer of 1968, he was employed by Collins Radio Company, Cedar Rapids, Iowa where he was involved in computer design and simulation.

He is currently studying under a National Science Foundation Fellowship and doing consulting work for Telpar, Inc., Dallas, Texas.

He is a member of Eta Kappa Nu, Tau Beta Pi, Phi Kappa Phi and The Institute of Electrical and Electronic Engineers.

Upon completion of his degree, he will be working for Bell Telephone Laboratories, Columbus, Ohio.